VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

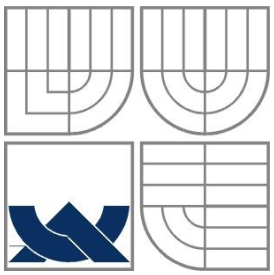
KOSIMULACE MIKROPROCESORU A PERIFERIÍ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

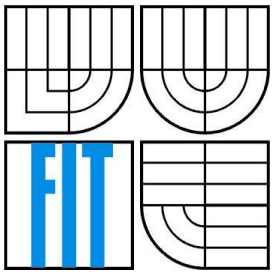
AUTOR PRÁCE
AUTHOR

Bc. Jan Frühbauer

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

KOSIMULACE MIKROPROCESORU A PERIFERIÍ

CO-SIMULATION OF MICROPROCESSOR AND PERIPHERAL DEVICES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

BC. JAN FRÜHBAUER

VEDOUCÍ PRÁCE
SUPERVISOR

PROF. ING. TOMÁŠ HRUŠKA, CSC.

BRNO 2012

Abstrakt

Cílem diplomové práce bylo analyzovat různé typy kosimulačních technik a navrhnout začlenění těchto technik do simulační platformy projektu Lissom.
První část práce ukazuje možnosti externích rozhraní simulačních platforem jazyků VHDL, Verilog a SystemVerilog a nástroje Matlab. Druhá část se věnuje návrhu a popisu implementace synchronizace simulační platformy projektu Lissom pomocí analyzovaných rozhraní s jinými simulačními platformami. V poslední části je popsáno testování implementovaného řešení a zhodnocení výsledků.

Abstract

The aim of this thesis is to analyze various kinds of co-simulation techniques and to design integration these techniques into Lissom simulation platform.
The first section of this thesis shows capabilities of external interfaces of simulation platforms for HDL languages VHDL, Verilog and SystemVerilog and of tool Matlab. The second section deals with design of synchronization mechanism among Lissom simulation platform and others simulation platforms using mentioned external interfaces. In the last part the testing of implemented solutions and evaluation of results is described.

Klíčová slova

Kosimulace, VHDL FLI, Verilog PLI, SystemVerilog DPI, Matlab, externí rozhraní, synchronizace

Keywords

Co-simulation, VHDL FLI, Verilog PLI, SystemVerilog DPI, Matlab, external interface, synchronization

Citace

Frühbauer Jan: Kosimulace mikroprocesoru a periferií, Diplomová práce, Brno, FIT VUT v Brně, 2012

Kosimulace mikroprocesoru a periferií

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením prof. Ing. Tomáše Hrušky, CSc.
Další informace mi poskytl Ing. Zdeněk Přikryl, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jan Frühbauer
23.5.2012

Poděkování

Děkuji vedoucímu práce prof. Ing. Tomáši Hruškovi, CSc. a konzultantovi Ing. Zdeňku Přikrylovi, Ph.D. za vedení mé diplomové práce, cenné rady a připomínky a za čas, který mi věnovali.
Dále děkuji mým rodičům a Haně Cinkové za podporu nejen během psaní diplomové práce, ale i během celého studia.

Obsah

1 Úvod

Zadání této práce mi bylo nabídnuto od kolektivu projektu Lissom. Projekt Lissom vyvíjí nástroj s názvem Codasip pro HW/SW co-design, jehož součástí je simulační platforma. Nástroj Codasip umožňuje navrhnout architekturu jádra procesoru a k tomuto návrhu vytvořit simulátor. Pro odhalení všech možných chyb návrhu však nestačí simulace pouze s jádrem procesoru, je potřeba provádět simulace s periferiemi, které se k jádru mohou připojovat. Připojení periferií pro účely simulování je možné použít kosimulaci. Kosimulace je současná simulace dvou či více simulačních platforem s využitím společného rozhraní mezi nimi. Mým úkolem tedy bylo začlenit do simulační platformy Codasip rozhraní pro kosimulaci s jinými simulačními platformami. Analyzoval jsem různé druhy externích rozhraní nabízené jinými simulačními platformami a na základě této analýzy jsem pak navrhnul začlenění těchto rozhraní do simulační platformy nástroje projektu Lissom.

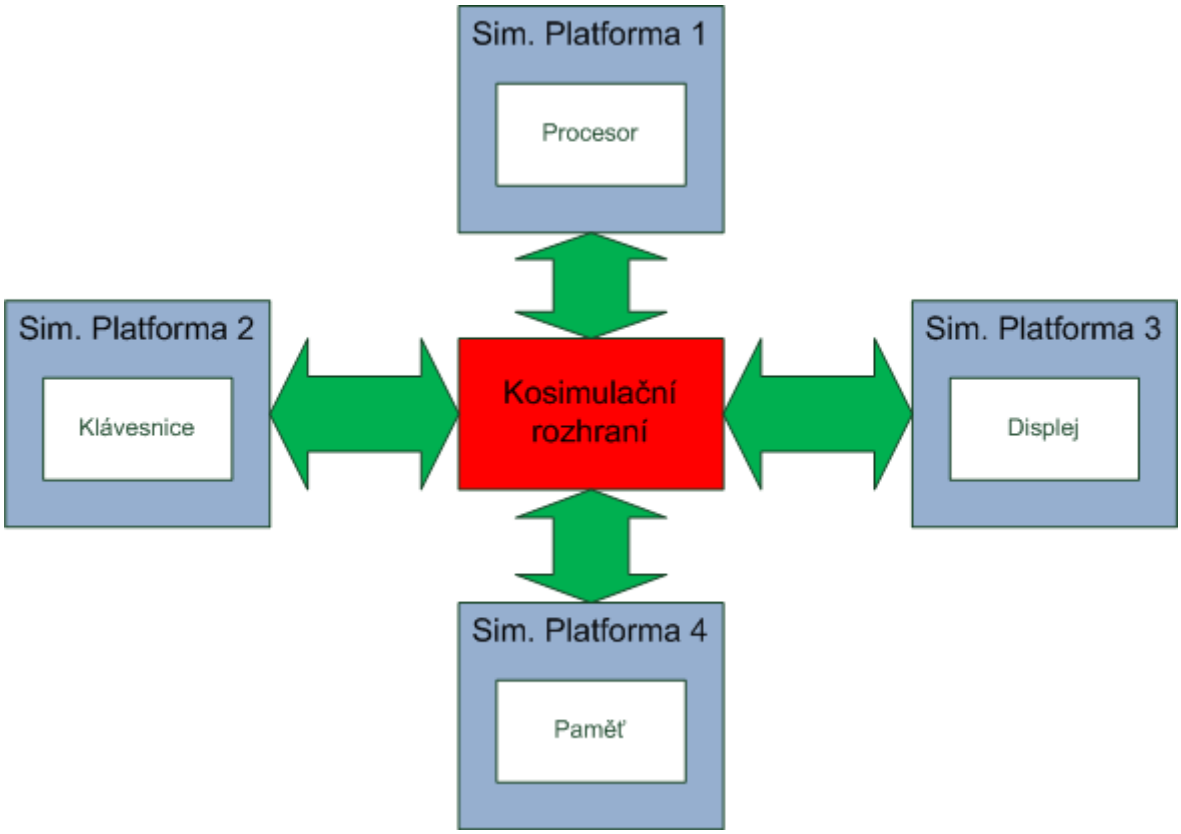
Pro analýzu byla vybrána externí rozhraní některých jazyků pro popis hardwaru. Simulační platforma VHDL nabízí externí rozhraní *Foreign Language Interface* (FLI) pro připojení sdílených knihoven psaných v jazyce C/C++ do popisu architektury modelu. Verilog má velmi podobné rozhraní *Programming Language Interface* (PLI), ale nabízí ještě větší možnosti přístupu k vnitřním datovým strukturám simulační platformy než FLI. Posledním jazykem pro popis hardwaru, který byl podroben analýze, je SystemVerilog. Protože je SystemVerilog pouze nástavbou k Verilogu, lze i u něj použít rozhraní PLI. Navíc však přináší nové rozhraní *Direct Programming Interface* (DPI), díky němuž je možné volat funkce jazyka C/C++ přímo ze SystemVerilogu a naopak funkce a úlohy ze SystemVerilogu volat v jazyce C/C++. Nakonec byla analyzována aplikace Matlab, která nabízí velké množství externích rozhraní pro C/C++, Fortran, Javu, .NET, webové služby či sériový port.

Na základě analýzy pak byla vybrána některá z těchto externích rozhraní a druhá část této práce uvádí návrh jejich začlenění k simulační platformě projektu Lissom. Návrh se zaměřuje na synchronizaci simulačních platforem a přenos informace mezi nimi.

Navržené řešení pro tato vybraná rozhraní jsem jako praktickou část svojí práce začlenil do nástroje Codasip. V kapitole o implementaci je popsán postup implementace s důrazem na zajímavé konstrukce. Poslední část této práce je pak zaměřena na testování implementovaného řešení na vhodném modelu mikroprocesoru a popis dosažených výsledků.

2 Kosimulace

V poslední době se využívají elektronické systémy v čím dál větší míře a je kladen velký důraz na zvyšování funkcionality zabudované v jednom čipu. Kvůli řešení problému rovnováhy mezi cenou zařízení a jeho flexibilitou se začal uplatňovat návrh hardwaru a softwaru současně, tzv. *HW/SW co-design*. Návrh takovýchto systémů je ale velmi komplexní, protože na čipu není pouze procesor, ale více zařízení, a z toho důvodu vystoupila do popředí celého procesu návrhu kosimulace, která umožňuje simulovat celý systém. Různé části systému jsou simulovány různými nástroji. Nicméně všechny tyto druhy kosimulace mají společný základ a to vytvoření mostu mezi simulačními platformami. Schéma kosimulace ukazuje obrázek č. 1.



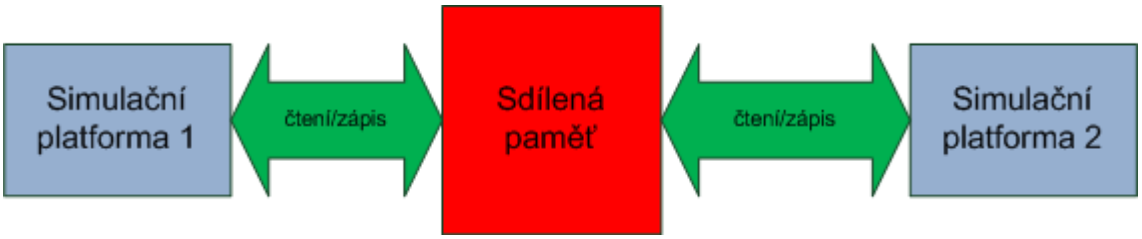
1 Schéma kosimulace

2.1 Komunikace mezi simulačními platformami

Nejdůležitějším prvkem kosimulace je ustavení komunikačního kanálu mezi simulačními platformami. Pomocí tohoto kanálu se potom synchronizují obě simulační platformy a předávají si data. Jakým způsobem spolu simulační platformy komunikují, závisí na platformě, na které jsou simulační platformy spouštěny. Vždy se však používají běžné principy zavedené pro komunikaci

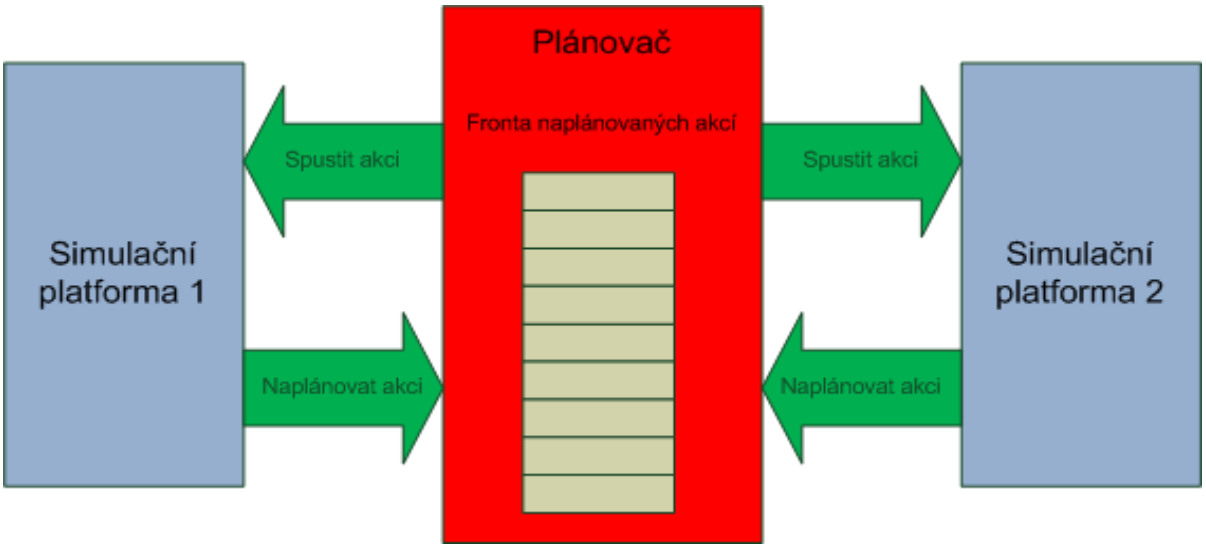
mezi aplikacemi a procesy (např. roury, TCP/IP spojení). Problémem při přenášení dat mezi simulačními platformami jsou datové typy těchto dat. Z toho důvodu velké množství jazyků nabízí knihovny externího rozhraní, ve kterých je převod interních datových struktur do reprezentace datových typů jiného jazyka.

Pomocí komunikačního kanálu dochází také k synchronizaci obou simulačních platforem. Pro synchronizaci se většinou používá model sdílené paměti (obrázek č. 2) s mechanizmem *čti-modifikuj-zapiš*. Přístup ke sdílené paměti je pak chráněn proti současnému přístupu obou simulačních platforem. Z toho důvodu pak dochází k blokování jedné simulační platformy druhou. Proto dochází ke sčítání dob simulací a tedy celkovému prodloužení kosimulace.



2 Kosimulace pomocí sdílené paměti

Další možností pro řízení kosimulace je plánování (obrázek č. 3). Zde je zaveden jeden globální čas a lokální časy po simulační platformy. Lokální čas je pak srovnáván s globálním a simulační platforma, která má lokální čas rovný globálnímu je aktivována. Řízení komunikace je pak dáno tím, že žádný z lokálních časů nesmí být větší než ten globální. Předchozí text vychází z [1].



3 Kosimulace pomocí plánovače akcí

3 Jazyky pro popis hardwaru a architektury

V nástroji Codasip se mohou procesory modelovat pomocí jazyka CodAL. Tento jazyk spadá do kategorie *jazyků pro popis softwarové architektury* (ADL). Softwarová architektura popisuje strukturu a chování softwarových systému a ne-softwarových prvků, se kterými tento systém spolupracuje. Systém je tedy reprezentován softwarovými komponentami, jejich propojením a způsobem komunikace mezi nimi. Jazyk pro popis architektury je formální jazyk pro modelování softwarových architektur na konceptuální úrovni. Hlavní výhodou použití těchto jazyků je možnost formálně specifikovat architekturu a poté ji analyzovat a případně verifikovat [2].

Jazyk pro popis architektury tedy musí být schopen reprezentovat komponenty, ať už primitivní nebo složené z jiných komponent, i s jejich rozhraním a implementací. Dále musí být schopen modelovat spojení mezi jednotlivými komponentami společně s komunikačním protokolem na těchto spojích. Toto vše musí být schopen zprostředkovat na vyšším stupni abstrakce a musí podporovat zapouzdření jednotlivých komponent [3].

Jelikož nástroj Codasip umožňuje pouze modelování procesorů, musí se periferie modelovat v jiném nástroji a pomocí kosimulačního rozhraní je připojit k procesoru. Pro modelování periferií se nabízí hlavně *jazyky pro popis hardwaru* (HDL). Tyto jazyky umožňují formálně popsat elektronické obvody (převážně číslicové). Hlavní výhodou těchto jazyků je popis funkce hardwaru nezávisle na jeho implementaci. Na rozdíl od softwarových programovacích jazyků obsahuje syntaxe HDL jazyků notace pro vyjádření času a souběžného provádění, což je při modelování hardwaru nezbytně nutné.

Popis obvodů pomocí HDL jazyků může být buď strukturální, nebo behaviorální. Strukturální popis obvodu je v podstatě textová reprezentace grafického schématu. Definují se jednotlivé součásti obvodu a jejich propojení do jednoho celku. Behaviorální popis na druhou stranu neobsahuje vůbec žádné informace o struktuře obvodu, ale popisuje pouze, co který obvod dělá [4].

4 Analýza rozhraní simulačních platforem

4.1 VHDL FLI

VHDL patří mezi jazyky pro popis hardwaru. Základním prvkem návrhu je entita. Entita komunikuje se svým okolím prostřednictvím portů. Popis chování entity se definuje v její architektuře, která je vůči okolí skrytá. Entity se dále mohou stát komponentou jiné entity a tím lze docílit hierarchického uspořádání [5].

FLI rutiny jsou funkce psané v programovacím jazyce C, které umožňují přístup k HDL simulační platformě *vsim*. Uživatel může použít tyto funkce k napsání aplikace, která může procházet jednotlivými komponentami návrhu, číst nebo modifikovat hodnoty objektů návrhu, získávat informace o simulaci a do určité míry ji i řídit.

4.1.1 Deklarace cizí architektury

Pro použití FLI se musí nejprve ve VHDL kódu vytvořit entita s architekturou s cizími parametry. Deklarace této architektury je následující:

```
architerture a of model is
    attribute  foreign of a: architekture is "init_funkce objekt.so; parametr;"
```

[6]

Init_funkce je inicializační funkce této architektury, která se vykoná při elaboraci. Objekt.so je objektový kód, ve kterém je popsána cizí architektura. Parametr je volitelný řetězec, který je předán inicializační funkci při elaboraci. Důležité je, že jakmile je jednou architektura deklarována jako cizí, tak se v ní již nemůže naprogramovat funkcionalita pomocí kódu VHDL. Veškerá funkcionalita se musí programovat v jazyce C/C++ s využitím rozhraní FLI. Jestliže by se v cizí architektuře nějaký kód VHDL objevil, tak je ignorován.

4.1.2 Inicializační funkce

Každá cizí architektura musí mít definovanou inicializační funkci, která slouží jako vstupní bod k modelu napsaném v jazyce C/C++. Tato funkce je volána při elaboraci a většinou slouží k alokaci paměti pro proměnné modelu, registrování *callback* funkcí, připojování signálů k portům, vytváření ovladačů těchto signálů, vytváření procesů a nastavení jejich citlivosti na dané signály.

Deklarace inicializační funkce:

```
init_funkce(mtiRegionIdT region, char *param, mtiInterfaceListT *generics, mtiInterfaceListT *ports)
```

[6]

Parametr region obsahuje identifikaci regionu, v rámci kterého je vytvořena instance cizí architektury, param je řetězec parametr z deklarace cizích parametrů architektury, generics je seznam proměnných typu *generic* dané instance a ports je seznam portů dané instance.

4.1.3 Cizí podprogramy

Další možností jak použít jazyk C/C++ ve VHDL jsou cizí podprogramy. Cizí podprogramy jsou VHDL funkce naprogramované v jazyku C/C++. Do kódu VHDL se začleňují podobně jako cizí architektura:

```
procedure proc();
    attribute foreign of proc : procedure is "c_proc objekt.so";
```

[6]

Parametr c_proc je název funkce v jazyce C/C++, která se vykoná, když bude v kódu VHDL zavolána funkce proc, objekt.so je objektový kód obsahující funkci c_proc.

4.1.4 Možnosti použití FLI

Práce s regiony

Každý region je jednoznačně identifikován pomocí identifikátoru a případně jména. Identifikátor je číselná hodnota, kterou pro identifikaci používá simulační platforma, kdežto jméno je řetězec, který je zaveden hlavně pro použití uživatelem. Pomocí funkcí FLI lze ze jména regionu určit identifikaci a naopak. Regiony se seskupují do hierarchické struktury návrhu. FLI umožňuje procházet tuto hierarchickou strukturu regionů. Hierarchii je možné procházet oběma směry. U každého regionu je totiž možné určit jeho synovské i rodičovské regiony a další regiony na stejné úrovni návrhu. Dále lze u regionu zjistit, zda je to VHDL nebo Verilog region. Další funkcí FLI je vytváření nových regionů a jejich začlenění do hierarchické struktury.

Práce s procesy

Každý proces je identifikován identifikátorem a případně jménem. FLI umožňuje vytvářet nové procesy v rámci cizí architektury. Je možné vytvořit proces s danou prioritou, která určí pořadí provádění procesů, pokud jsou naplánovány na stejný simulační čas. Dále lze získat identifikátory

procesů v jednotlivých regionech a u procesu lze zjistit, do kterého regionu spadají. U procesu se pak stejně jako ve VHDL může nastavit citlivost na změnu nebo aktivitu nějakého signálu. Provedení procesu může být také naplánováno na určitý simulační čas.

Práce se signály

Každý signál je identifikován identifikátorem a případně jménem. Pomocí FLI lze vytvářet nové signály, hledat signál daného jména, určit první signál v daném regionu a další signály na stejné úrovni hierarchie návrhu. Dále je možno zjistit, jaký signál je mapován na daný port. Jsou dostupné funkce pro zjištění jména signálu nebo jména složek složeného signálu, pro zjištění hodnoty a směru signálu nebo jeho složek a nakonec pro nastavení hodnoty daného signálu. Každému signálu lze přiřadit libovolný počet ovladačů, které řídí hodnoty signálů. U ovladače lze naplánovat změnu hodnoty signálu, zjistit jeho současnou hodnotu nebo jeho název. U signálu je potom možné rozpoznat, jestli k němu náleží nějaký ovladač nebo ne.

Práce s proměnnými

Proměnné na rozdíl od signálů nelze v cizí architektuře vytvářet. Lze pouze používat proměnné deklarované ve VHDL procesu. Pro práci s proměnnými jsou k dispozici stejné funkce jako pro signály, tedy zjištění jména nebo hodnoty proměnné nebo složky složené proměnné (například proměnná složená z více bitů) a nastavení hodnoty proměnné. Proměnné se na rozdíl od signálů nepřiřazují k regionům, ale k procesům. U každé proměnné lze navíc získat přímý ukazatel do paměti na místo, kde je hodnota proměnné uložena a lze tedy modifikovat hodnotu proměnné přímým zápisem do paměti bez použití speciální funkce FLI.

Práce s datovými typy

Pomocí FLI lze vytvářet vlastní datové typy – pole s danými mezemi a datovým typem položek, výčtový typ s danými literály, skalární typ s danými hranicemi a nový identifikátor datového typu *real* a *time*. U ohraničených typů lze zjistit jejich pravou, levou, horní a dolní hranici a u každého typu lze zjistit směr a velikost. U pole je možné určit typ položek, u záznamu počet položek a u výčtového typu seznam všech literálů.

Práce s callback funkcemi

FLI nabízí možnost registrace a zrušení registrace callback funkcí. Tyto funkce jsou volány kdykoliv dojde ke specifické události, pro kterou jsou registrovány.

Události, pro které lze registrovat callback funkce:

- změna prostředí simulační platformy.
- dokončení elaborace.
- ukončení simulační platformy.
- restart simulační platformy.
- obnovení simulační platformy.
- ukončení obnovení simulační platformy.
- záloha simulační platformy.

- změna stavu simulační platformy.
- vstupní nebo výstupní soubor, socket nebo roura připravena ke čtení nebo zápisu.

Práce s pamětí

FLI nabízí možnost alokovat, realokovat a uvolňovat místo v paměti simulační platformy. Jinak lze samozřejmě používat klasické funkce jazyka C/C++ pro správu paměti programu.

Zálohování a obnova dat

V cizí architektuře lze libovolně zálohovat a obnovovat data. FLI nabízí funkce pro uložení a obnovu bloku dat, jednoho bytu, řetězce a proměnných typu *short* a *long*. Vždy je ale nutné zachovat pořadí zálohování a obnovy. Při obnově ze zálohy je také potřeba vždy obnovit procesy vytvořené v rámci cizí architektury. Při obnově se totiž může změnit umístění funkcí procesů.

Práce s časem a naplánovanými událostmi

FLI umožňuje přístup k simulačnímu času. Simulační čas je reprezentován 64-bitovým číslem. Pomocí FLI lze získat buď celý 64-bitový čas, nebo jen jeho horní nebo dolní polovinu. Dále je možné získat počet událostí v současném simulačním čase. Další informací, dostupnou pomocí FLI, je jednotka času simulační platformy nebo koncový čas současného běhu simulace. Kromě současného simulačního času lze také získat čas další události v cizí architektuře a čas další události z VHDL procesu.

Práce s příkazy simulační platformy

V cizí architektuře je možné vyvolat provedení jakéhokoli příkazu v simulační platformě kromě příkazů, které mění stav simulační platformy. Dále uživatel může přidat k simulační platformě vlastní příkaz a funkci, která se po zavolání příkazu vykoná. FLI také nabízí funkce pro žádost o pozastavení simulace, zastavení simulace kvůli kritické chybě a uzavření simulační platformy.

Vstupně-výstupní komunikace

FLI nabízí dvě funkce pro výpis zpráv do okna simulační platformy. Jedna vypíše prostý řetězec na výstup a druhá formátovaný řetězec (funkčnost je stejná jako u funkce *printf* jazyka C). Dále nabízí FLI funkci pro získání informací od uživatele. Vypíše do okna simulační platformy výzvu k zadání požadované informace a odpověď předá do programu.

Předchozí text vychází z [6].

4.2 Verilog PLI

Verilog je další z rodiny jazyků pro popis hardwaru. Modely psané ve Verilogu mají velmi podobnou strukturu jako modely VHDL. Základním prvkem modelu je modul, který s okolí komunikuje pomocí portů stejně jako entita ve VHDL. V modulu se pak definuje i jeho vnitřní struktura a chování. Moduly se mohou také vnořovat do sebe a tvořit hierarchický návrh [7].

Verilog Programming Language Interface, označovaný zkráceně Verilog PLI, je procedurální rozhraní pro Verilog simulační platformy. Umožňuje rozšířit možnosti simulační platformy voláním programů psaných v jazyce C/C++.

Použití PLI:

1. Vytváření *bus-functional* modelů – psaní těchto modelů v jazyce C/C++ může sloužit k optimalizaci simulace nebo k ochraně duševního vlastnictví.
2. Přístup ke knihovnám jazyka C/C++ – programy psané ve Verilogu mohou použít knihovny jazyka C/C++, kterých je velké množství. Pomocí PLI se mohou argumenty z Verilogu předat funkcím z knihoven a výsledek vrátit zpět do Verilogu.
3. Načítání dat ze vstupních souborů – aplikace PLI může číst data ze vstupního souboru, která předává simulaci modelu ve Verilogu.
4. Výpočet zpoždění – výrobci ASIC a FPGA mohou poskytovat program pro výpočet zpoždění. Na základě aktuálních podmínek simulace dokáže vypočítat zpoždění jednotlivých hradel. Pomocí PLI se pak na základě výpočtu mohou upravit datové struktury simulace, takže se celá simulace zpřesní.
5. Zobrazení výstupu simulace – mohou se přidávat nové způsoby zobrazení výsledku, které jsou například přehlednější nebo uživatelsky přívětivější.
6. Kosimulace – PLI může být využito jako komunikační kanál mezi simulační platformou Verilogu a jiným typem simulační platformy.
7. Nástroje pro ladění – debugger poskytovaný v simulační platformě nemusí poskytovat veškeré informace, které by byly zapotřebí. Pomocí PLI lze vytvořit debugger, který má přístup ke všem datům simulační platformy Verilog v rámci simulace.
8. Analýza simulace – pomocí PLI se může určit, co se přesně dělo v některém čase simulace s prvky návrhu, určit, které příkazy Verilogu byly při simulaci vykonány atd.

4.2.1 Základní kroky pro vytvoření PLI aplikace

Vytváření každé aplikace pomocí knihoven PLI se skládá z několika kroků, které musí být vždy splněny:

1. Definice jména systémové úlohy nebo systémové funkce pro novou aplikaci
2. Vytvoření funkce *calltf* v jazyce C/C++, která bude vykonána, když simulační platforma narazí na jméno systémové úlohy nebo funkce
3. Registrace jména systémové úlohy nebo funkce a s ní asociované funkce
4. Překlad zdrojového souboru v jazyce C/C++ a přilinkování objektového souboru k simulační platformě

4.2.2 Systémové úlohy a funkce

V jazyce Verilog jsou systémové úlohy a funkce příkazy, které jsou vykonány simulační platformou. Jejich názvy začínají vždy znakem '$'. Systémové úlohy se používají jako samostatné příkazy programu, kdežto systémové funkce se používají stejně jako programové funkce, které vrací hodnotu, hlavně ve výrazech.

Standard jazyka Verilog definuje tři typy systémových úloh a funkcí:

1. Standardní množinu vestavěných systémových úloh a funkcí – všechny simulační platformy Verilogu musí obsahovat tyto úlohy a funkce.
2. Systémové úlohy a funkce specifické pro simulační platformu – úlohy a funkce, které definoval tvůrce simulační platformy. Každá simulační platforma může mít jinou množinu těchto úloh a funkcí.
3. Systémové úlohy a funkce definované uživatelem – tyto úlohy a funkce jsou právě tvořeny pomocí PLI.

Pokud uživatel definuje jméno své vlastní úlohy nebo funkce stejně jako nějaká vestavěná úloha nebo funkce, tak je vestavěná úloha nebo funkce nahrazena uživatelovou.

4.2.3 Kompilace a linkování PLI aplikace

Standard PLI neobsahuje směrnice pro kompilaci a linkování PLI aplikací. Vše je závislé pouze na aplikační platformě a simulační platformě Verilogu. Proces kompilace a linkování bývá předepsán výrobcem simulační platformy.

4.2.4 Standard PLI 1.0

Tento standard obsahuje dvě knihovny – TF a ACC. TF (**T**ask/**F**unction) je nejstarší knihovnou a jak napovídá název, tak poskytuje hlavně rutiny pro přístup k informacím o argumentech uživatelsky definovaným úlohám a funkcím. ACC (**Acc**ess) je pak knihovna určená především pro přístup k informacím simulační platformy.

4.2.4.1 Typy PLI rutin verze 1.0

PLI standart definuje několik typů rutin, které mohou být asociovány se systémovou úlohou nebo funkcí. Typ rutiny určuje, kdy simulační platforma tuto rutinu vykoná.

1. Rutiny *calltf* – jsou vykonávány při běhu simulace, kdykoli simulační platforma dojde v programu na místo volání systémové úlohy nebo funkce.
2. Rutiny *checktf* – jsou vykonány před spuštěním samotné simulace - to znamená před časem simulace 0. Tyto rutiny většinou kontrolují, zda je systémová úloha nebo funkce správně použita (správný počet a typ parametrů). Dále je vhodné využít tuto rutinu ke zlepšení

rychlosti simulace. Mohou se v ní udělat operace, které se opakují v každé rutině *calltf*. Jelikož je rutina *checktf* volána na rozdíl od *calltf* pouze jednou, nespotřebovávají pak tyto operace tolik procesorového času. Musí se ale brát zřetel na ten fakt, že tyto rutiny jsou prováděny před časem 0 a tedy proměnným ještě nemusely být přiřazeny počáteční hodnoty. Není tedy vhodné jakkoli používat proměnné.

3. Rutiny *sizetf* – tyto rutiny se používají pro systémové funkce, které mají proměnnou velikost návratové hodnoty. Protože tyto funkce vrací uživatelem definovaný počet bitů, kompilátor simulační platformy se musí nějak dozvědět, kolik bitů má v daném případě očekávat, aby mohl správně zkompilovat příkaz, ve kterém je daná funkce volána. Právě rutiny *sizetf* slouží k tomuto účelu. Jsou volány před časem simulace 0 a vrací počet bitů návratové hodnoty systémové funkce.
4. Rutiny *misctf* – tyto rutiny jsou provedeny jako reakce na nějakou událost simulační platformy. Události, na které je možno reagovat, jsou konec elaborace, konec simulace, vstup do ladícího režimu, změna hodnoty argumentu uživatelem definované systémové úlohy, konec jednoho simulačního kroku a dosažení specifického simulačního kroku. K těmto standardem definovaným událostem může výrobce simulační platformy přidat svoje vlastní.

Všechny PLI rutiny jsou funkce jazyka C/C++ a mají tedy své vstupy a návratové hodnoty.

```
int rutina(int user_data, int reason);
```

Prvním vstupním argumentem rutin *calltf, checktf, sizetf* a *misctf* je ukazatel na uživatelská data, která byla specifikována při registraci systémové úlohy nebo funkce, a druhým je důvod, proč jsou volány. Druhý vstupní argument je používán pouze v rutině *misctf,* v ostatních rutinách je ignorován. Rutina *misctf* má navíc ještě jeden argument, který je použit, jestliže je rutina zaregistrována jako reakce na změnu argumentu uživatelem definované systémové úlohy. Pak tento argument obsahuje informaci, který argument se změnil. Všechny PLI rutiny mají návratový typ *integer*. Ovšem používána je pouze návratová hodnota rutiny *sizetf*. Ostatní návratové hodnoty simulační platforma ignoruje.

4.2.4.2 Připojení TF/ACC aplikace k simulační platformě Verilogu

Mechanizmus rozhraní pro TF/ACC aplikace definuje ve starším standardu pouze co by měla simulační platforma Verilog poskytovat pro připojení těchto aplikací. Nespecifikuje však, jak by se mělo toto rozhraní implementovat. Závisí tedy na každé simulační platformě, jakou implementaci zvolí. Většina simulačních platforem ale používá strukturu *s_tfcell*, která obsahuje položky pro definici připojení aplikace k simulační platformě. Simulační platforma potom načítá informace o

všech připojených aplikacích z pole těchto struktur. A protože tuto strukturu využívá naprostá většina simulačních platforem, dá se označit za standard. Do této struktury se uloží následující informace:

- název systémové úlohy nebo funkce,
- typ PLI aplikace (úloha nebo funkce),
- libovolná uživatelská data, která budou předány rutinám *calltf, checktf, sizetf* a *misctf,*
- ukazatele na funkce jazyka C/C++, které budou volány simulační platformou pro rutiny *calltf, checktf, sizetf* a *misctf.*

4.2.4.3 Rutiny knihovny TF

Knihovna TF nabízí celkem 104 rutin převážně pro přístup k informacím o argumentech systémové úlohy nebo funkce, dále pak pro řízení simulace nebo výpis hlášení. Hlavní výhodou použití této knihovny je výkon. Jelikož knihovna TF nenabízí přístup do vnitřních datových struktur simulace, takže simulační platforma může efektivněji optimalizovat tyto struktury.

Argumenty systémové úlohy nebo funkce

Knihovna TF čísluje argumenty systémových úloh nebo funkcí od jedničky v pořadí zleva doprava. Pomocí pořadových čísel lze číst jednotlivé argumenty. Lze získat celkový počet argumentů, dále typ každého z nich a jejich velikost v bitech. Jelikož jedna systémová úloha nebo funkce se může vyskytovat ve více instancích, tak knihovna TF nabízí funkci pro získání ukazatele na danou instanci. Tímto způsobem lze pak například přistupovat i k argumentům jiných instancí. Pokud je argumentem systémové úlohy nebo funkce řetězec, tak se při přečtení jeho hodnoty pomocí TF uloží řetězec do speciálního bufferu knihovny. Každým novým přečtením řetězce se však buffer přepisuje a proto je nutné si řetězec zkopírovat na nově alokované místo v paměti aplikace, pokud se má dále používat.

Řízení simulace

V knihovně TF jsou dvě rutiny pro řízení simulace. První rutina přepne simulační platformu do interaktivního ladícího prostředí a druhá slouží pro ukončení simulace.

Výpis hlášení

Knihovna TF nabízí několik rutin pro výpis různých druhů hlášení. Existují speciální rutiny pro výpis varování, chyb, nebo obyčejného textu na výstupní kanál simulační platformy nebo do souboru otevřeného v rámci kódu Verilogu. Všechny tyto rutiny mají syntaxi stejnou s funkcí *printf* jazyka C. Všechny rutiny ale mají omezený počet argumentů.

Pracovní prostor

Každá instance PLI aplikace má implicitně alokovaný pracovní prostor pro uložení ukazatele na libovolná uživatelská data. Tento pracovní prostor je pak sdílen všemi PLI rutinami dané instance PLI aplikace. Knihovna TF pak obsahuje rutiny pro uložení a načtení ukazatele z pracovního prostoru.

Simulační čas

Pomocí rutin knihovny TF lze přistupovat k simulačnímu času. Lze získat aktuální simulační čas v různých formátech, dále také simulační čas, na který je naplánována nejbližší simulační událost. Existují i rutiny pro zjištění aktuálního nastavení jednotky času a přesnosti.

Ostatní rutiny

Knihovna TF obsahuje několik užitečných rutin pro práci s Verilog modelem a simulací. Jde například o matematické operace pro 64-bitové hodnoty nebo získání modulu, ve kterém se nachází instance příslušné systémové úlohy nebo funkce.

4.2.4.4 Rutiny knihovny ACC

Knihovna ACC obsahuje 103 rutin určených především pro přístup k interním datovým strukturám simulační platformy, jako jsou informace o modulech, signálech, portech atd. Rutiny této knihovny zachází s konstrukcemi Verilogu jako s objekty. Nemůže však přímo přistupovat k objektům, ale má k dispozici pouze ukazatele na strukturu, která obsahuje informace o objektu.

Naprostá většina rutin z knihovny ACC je určena právě pro práci s ukazateli na datové struktury obsahující informace o objektech simulace. Například pro samotnou lokalizaci objektů a získání informací o nich je vytvořeno 45 rutin. Ostatní rutiny jsou pak určeny pro čtení a modifikaci informací o jednotlivých objektech. Všechny rutiny jsou přeneseny v obdobné formě i do novějšího standardu pouze s tím rozdílem, že ve standardu PLI 2.0 je jako objekt bráno více konstrukcí Verilogu, takže nabízí větší možnosti.

Kromě přístupu k informacím o objektu pak nabízí ještě jedno rozšíření oproti knihovně TF. Dokáže zaregistrovat funkci, která se má spustit při změně hodnoty nějakého objektu. V podstatě jde jen o rozšíření funkčnosti PLI rutiny *misctf*.

4.2.5 Standard PLI 2.0

Standard PLI 2.0 kompletně nahradil starší standard PLI 1.0. Nicméně starší standard je stále podporován ve většině Verilog simulačních platformách z důvodů zpětné kompatibility. Knihovny TF a ACC byly nahrazeny knihovnou VPI, která je mnohem jednodušší na používání při zachování veškeré funkcionality starších knihoven.

4.2.5.1 Typy PLI rutin verze 2.0

PLI standart definuje několik typů rutin, které mohou být asociovány se systémovou úlohou nebo funkcí. Typ rutiny určuje, kdy simulační platforma tuto rutinu vykoná.

1. rutiny *calltf* – odpovídá rutině *calltf* ze standardu PLI 1.0.
2. rutiny *compiletf* – odpovídá rutině *checktf* ze standardu PLI 1.0
3. rutiny *sizetf* – odpovídá rutině *sizetf* ze standardu PLI 1.0

4. rutiny *callback* – tyto rutiny jsou provedeny jako reakce na nějakou událost simulační platformy. Jde například o začátek nebo konec simulace, změna hodnoty signálu, vstup do ladícího režimu atd.

   Všechny PLI rutiny jsou funkce jazyka C/C++ a mají tedy své vstupy a návratové hodnoty.

```
PLI_INT32 rutina(PLI_INT8 *user_data);
```

Vstupním atributem rutin *calltf*, *compiletf* a *sizetf* je ukazatel na uživatelská data, která byla specifikována při registraci systémové úlohy nebo funkce. Vstupem rutiny *callback* je ukazatel na strukturu obsahující informace o daném *callbacku*. Všechny PLI rutiny mají návratový typ *int* (PLI_INT32 je pouze definice nového jména pro typ int). Ovšem používána je pouze návratová hodnota rutiny *sizetf*. Ostatní návratové hodnoty simulační platforma ignoruje.

4.2.5.2 Připojení VPI aplikace k simulační platformě Verilogu

Standard PLI poskytuje prostředky k vytvoření asociace mezi systémovou úlohou nebo funkcí a PLI aplikací. Pro vytvoření této asociace se musí napsat další funkce v jazyce C/C++. V této funkci se alokuje místo pro strukturu, která byla vytvořena jako součást knihovny VPI přímo pro registraci PLI aplikace. Do této struktury se uloží následující informace:

- název systémové úlohy nebo funkce,
- typ PLI aplikace (úloha nebo funkce),
- pro systémovou funkci její návratový typ,
- libovolná uživatelská data, která budou předány rutinám *calltf*, *compiletf* a *sizetf*,
- ukazatele na funkce jazyka C/C++, které budou volány simulační platformou pro rutiny *calltf*, *compiletf* a *sizetf*.

Po naplnění struktury už zbývá pouze zavolat VPI rutinu pro registraci této struktury v simulační platformě. Tímto je hotova registrační funkce. Nyní je zapotřebí upozornit simulační platformu na jméno této registrační funkce. Může se to dělat několika způsoby. Například se může použít speciální pole simulační platformy *vlog_startup_routines*, do kterého se uloží jména všech registračních funkcí.

4.2.5.3 Rutiny knihovny VPI

Knihovna VPI obsahuje 37 rutin. Na rozdíl od staršího standardu, kde bylo přes 200 rutin, jde o výrazné zpřehlednění. Rutiny zpracovávají konstrukce jazyka Verilog jako objekty. Rutiny pak dokážou najít objekty v datové struktuře simulace, číst informace o objektech a měnit je.

Rutiny se dělí do pěti základních skupin:

1. Rutiny pro získání ukazatele na jeden specifický Verilog HDL objekt.

2. Iterační a skenovací rutiny pro získání ukazatelů na všechny objekty specifického typu.
3. Rutiny pro čtení informací o objektech.
4. Rutiny pro modifikaci informací o objektech.
5. Několik dalších rutin pro různé operace.

Rutiny pro práci s ukazateli

Ukazatel není ukazatelem v pravém slova smyslu. Neukazuje přímo na daný objekt, ale pouze na strukturu obsahující informace o daném objektu. Z toho vyplývá, že nelze porovnávat ukazatele běžným operátorem rovnosti z jazyka C/C++, protože dvě různé struktury mohou obsahovat informace o tomtéž objektu. Knihovna VPI nabízí rutinu speciálně určenou pro porovnávání ukazatelů na objekty. Všechny rutiny pro získání ukazatele mají dva argumenty. Prvním je typ cílového objektu, pro který chceme ukazatel získat a druhým je ukazatel na referenční objekt, v rámci kterého se hledaný objekt nachází. Rutiny se dělí podle toho, v jakém vztahu jsou cílový a referenční objekt. Buď mohou být ve vztahu jeden ku jednomu, nebo jeden ku mnoha. Pokud jsou ve vztahu jeden ku mnoha, tak výsledkem rutiny pro získání ukazatelů není jeden ukazatel, ale iterátor přes všechny cílové objekty.

Rutiny pro práci s vlastnostmi objektů

Každý Verilog objekt má jednu nebo více vlastností. Většina vlastností může být pomocí VPI rutin získána buď v podobě číselné, nebo řetězcové hodnoty.

Některé typy vlastností jsou:

- typ objektu – například modul, datový typ *net* nebo *reg*, nebo primitivní datový typ,
- jméno objektu – lze získat více typů jmen – lokální jméno, plné hierarchické jméno, název definice,
- logická hodnota objektu – je zachycena pomocí struktury, která udává formát hodnoty (jak se bude hodnota prezentovat v jazyce C/C++) a pak samotnou hodnotu.

Rutiny pro získání času

Simulační platforma Verilogu umožňuje určit každému modulu jinou jednotku simulačního času. Knihovna VPI nabízí rutiny pro získání času v jednotkách libovolného modulu nebo v jednotkách interního simulačního času.

Rutiny pro řízení simulace

Knihovna VPI nabízí rutinu pro předávání příkazů přímo simulační platformě. Samotný standard obsahuje definici pouze několika operací simulační platformy, které musí být možné zavolat z PLI aplikace. Každá simulační platforma si pak může libovolně definovat další rozšiřující operace. Základními operacemi jsou přechod do interaktivního ladícího režimu, ukončení a resetování simulace a změna interaktivního prostoru ladění.

Další rutiny

Knihovna VPI pak například nabízí rutinu pro výpis textu na výstup simulační platformy a do logovacího souboru. Pokud simulační platforma používá jako svůj výstup standardní výstup operačního systému, tak se mohou pro výpis použít i funkce jazyka C/C++. Dále nabízí rutinu pro získání informací o chybách, které se staly v rámci poslední volané VPI rutiny.

Předchozí text vychází z [8].

4.3 SystemVerilog DPI

SystemVerilog vznikl na základě jazyka Verilog. Má tedy všechny vlastnosti Verilogu. Tento jazyk byl definován hlavně z důvodu podpory verifikace modelů hardwaru. Oproti Verilogu přináší nové datové typy, procedurální příkazy, rozšířené možnosti použití úloh a funkcí atd.

DPI je rozhraní mezi SystemVerilogem a cizím programovacím jazykem. Skládá se ze dvou oddělených vrstev – vrstvy SystemVerilogu a vrstvy cizího programovacího jazyka. Tyto vrstvy jsou plně izolovány, takže pro vrstvu SystemVerilogu je jedno, jaký jazyk se použije na místě vrstvy cizího jazyka. Toto rozhraní vzniklo hlavně kvůli jednoduššímu propojení SystemVerilogu s jinými jazyky než jaké nabízí rozhraní PLI.

4.3.1 Úlohy a funkce

DPI umožňuje přímé volání funkcí mezi SystemVerilogem a cizím jazykem. Lze tedy volat funkce z cizího jazyka v SystemVerilogu. Tyto funkce se pak nazývají importované funkce. A dále je možné volat i funkce psané v SystémVerilogu v cizím jazyce. Pak se tyto funkce nazývají exportované. To samé jako pro funkce platí pro úlohy. Rozdíl v importované/exportované funkci a úloze spočívá jednak v tom, že funkce má nějakou návratovou hodnotu a úloha ne, a dále v tom, že úloha může spotřebovávat simulační čas, kdežto funkce musí být provedena v nulovém simulačním čase. Importované úlohy a funkce mohou mít libovolný počet formálních argumentů. Tyto argumenty se dělí do tří kategorií:

1. Vstupní – argumenty, které by neměli být měněny a pokud jsou změněny, tak se tyto změny nesmí projevit mimo úlohu nebo funkci.
2. Výstupní – argumenty, do kterých se může zapisovat hodnota, ale neměla by se číst jejich inicializační hodnota.
3. Vstupně-výstupní – kombinace obou předchozích, lze číst jejich inicializační hodnotu a může se do nich zapisovat nová hodnota.

Pokud se změní hodnota výstupního nebo vstupně-výstupního argumentu, tak simulační platforma SystemVerilogu je zodpovědná za propagaci této změny. Tato propagace se děje ihned po předání řízení z importované úlohy nebo funkce zpátky simulační platformě. Samotné volání importovaných úloh a funkcí je naprosto shodné s voláním funkcí SystemVerilogu.

4.3.2 Datové typy

Mezi dvěma vrstvami DPI lze předávat pouze proměnné datových typů z jazyka SystemVerilog. Vrstva cizího jazyka se tedy musí starat o převod vlastních datových typů na typy SystemVerilogu a naopak.

4.3.2.1 Návratové hodnoty funkcí

Jako návratové typy funkcí mohou být použity pouze malé hodnoty. Následující datové typy SystemVerilogu mohou být použity jako návratové:

- základní typy *void*, *byte*, *shortint*, *int*, *longint*, *real*, *shortreal*, *chandle* a *string*,
- zhuštěná bitová pole větší než 32 bitů,
- skalární typy *bit* a *logic*.

4.3.2.2 Datové typy formálních argumentů

Formální argumenty importovaných a exportovaných úloh a funkcí mohou nabývat více datových typů než návratové hodnoty funkcí. Jsou to tyto datové typy:

- základní typy *void*, *byte*, *shortint*, *int*, *longint*, *real*, *shortreal*, *chandle* a *string*,
- skalární typy *bit* a *logic*,
- zhuštěná jednodimenzionální pole typu *bit* a *logic*,
- výčtové typy interpretované jako typ asociovaný s daným výčtem,
- typy zkonstruované z předchozích pomocí *struct*, *typedef* a polí.

Pole, ať už zhuštěná nebo ne, mohou být použity také v otevřené formě. To znamená, že jedna nebo více dimenzí nemá specifikovaný rozsah. Otevřená pole pak umožňují použití generického kódu pro zpracování různých velikostí polí.

4.3.3 Speciální vlastnosti importovaných úloh a funkcí

Při definici importovaných úloh a funkcí lze specifikovat dvě speciální vlastnosti – *pure* a *context*.

4.3.3.1 Vlastnost pure

Vlastnost *pure* lze specifikovat pouze pro funkce s návratovou hodnotou (ne typu *void*), jejichž výstup závisí pouze na vstupních argumentech. Zároveň také nesmí provádět žádnou vedlejší činnost. Z toho vyplývá, že funkce nesmí mít žádné vstupní ani vstupně-výstupní argumenty, nesmí provádět žádné operace se soubory, nesmí přistupovat k proměnným prostředí, objektům operačního systému, sdílené paměti, socketům atd. Pokud funkce splňuje tyto požadavky a je definována jako *pure*, tak potom může kompilátor bezpečně odstranit volání této funkce, když návratová hodnota není použita nebo byla vypočítána pro dané vstupní argumenty již dříve a není potřeba ji znovu

přepočítávat. Musí se ale dávat pozor, aby opravdu splňovala všechny požadavky. Pokud by je nesplnila a přesto byla definována jako *pure*, tak může dojít k neočekávanému chování.

4.3.3.2 Vlastnost context

Pokud importovaná úloha nebo funkce vyžaduje, aby znala svůj kontext volání, musí se definovat pomocí vlastnosti *context*. SystemVerilog nepředává importovaným úlohám a funkcím kontext volání jejich instancí, dokud nejsou definovány jako *context*. Je to takto děláno z důvodů optimalizace, protože úlohy a funkce, které nejsou takto definovány, nemohou přistupovat k vnitřním datovým strukturám SystemVerilogu kromě skutečných argumentů volání a nebrání tedy v provádění optimalizace. Pokud se specifikuje vlastnost *context*, tak úloha nebo funkce může použít například knihovny PLI pro přístup k vnitřním objektům simulační platformy a libovolně s těmito objekty pracovat. Dále je umožněno volání exportovaných úloh a funkcí ze SystemVerilogu. Ovšem exportované úlohy mohou volat pouze importované úlohy a ne funkce. Je to proto, že exportované úlohy by mohly spotřebovávat simulační čas.

4.3.4 Deklarace importovaných úloh a funkcí

Každá importovaná úloha a funkce musí být v SystemVerilogu deklarována. Deklarace má následující formát:

import "DPI" [pure | context] [identifikátor =] prototyp;

[4]

kde identifikátor je název úlohy nebo funkce, pod kterým se bude používat v SystemVerilogu, a prototyp je prototyp importované úlohy nebo funkce. Pokud není specifikován identifikátor, tak se jako název vezme název úlohy nebo funkce z prototypu.

4.3.5 Deklarace exportovaných úloh a funkcí

Každá exportovaná úloha a funkce, která může být použita v cizím programu, musí být jako exportovaná deklarována. Ze SystemVerilogu mohou být exportovány všechny úlohy a funkce kromě těch, které jsou částmi tříd.

export "DPI" [identifikátor =]**function** function_identifikátor;
export "DPI" [identifikátor =] **task** task_identifikátor;

[4]

Deklarace má podobný formát jako u importovaných úloh a funkcí. Opět pokud není specifikován identifikátor, tak se jako název úlohy nebo funkce vezme identifikátor z definice.

4.3.6 Blokování DPI úloh a funkcí

V programu SystemVerilogu je možné zablokovat importované a exportované úlohy a funkce. Z toho důvodu by měla každá importovaná úloha a funkce implementovat jednoduchý protokol pro blokování. Díky tomuto protokolu je umožněno programu v cizím jazyce, aby uvolnil paměť, zavřel otevřené soubory atd. Jedinou možností pro importované úlohy a funkce jak zjistit, že simulační platforma je v blokovacím režimu, je ihned po volání exportované úlohy nebo funkce zavoláním funkce *svIsDisabledState()*. Protokol se skládá z následujících kroků:

1. Když volaná exportovaná úloha skončí kvůli blokování, musí se vrátit hodnota 1. Jinak se vrací 0.
2. Když volaná importovaná úloha skončí kvůli blokování, musí se vrátit hodnota 1. Jinak se vrací 0.
3. Než skončí importovaná funkce, která zjistila, že je blokována, musí zavolat funkci *svAckDisabledState()*.
4. Jakmile importovaná úloha nebo funkce zjistí, že se nachází v blokovaném režimu, tak je zakázáno další volání exportovaných úloh a funkcí.

4.3.7 Použití jazyka C/C++ jako cizího jazyka

Standart pro DPI definuje rozhraní mezi vrstvami pouze pro jediný jazyk a to jazyk C/C++. Jiné jazyky mohou DPI také využívat, ale ke každému jazyku by se muselo vytvořit vlastní rozhraní mezi vrstvami DPI.

DPI pro jazyk C je definováno pomocí dvou hlavičkových souborů. První z nich *svdpi.h* je základní a při použití pouze tohoto souboru je zaručena binární kompatibilita programů se všemi simulačními platformami. Jestliže je potřeba i druhý hlavičkový soubor *svdpi_src.h*, tak už jsou programy pouze zdrojově kompatibilní. Zdrojová kompatibilita je dána tím, že tento soubor obsahuje definice reprezentace zhuštěných polí, které jsou závislé na simulační platformě.

Předchozí text vychází z [9].

4.4 Rozhraní Matlabu

Matlab je nástroj pro vývoj algoritmů, analýzu dat, vizualizaci a numerické výpočty. Pomocí Matlabu je možné vyřešit technické výpočetní problémy rychleji než při použití tradičních programovacích jazyků [10].

4.4.1 Soubory MAT

Matlab nabízí pro export a import dat z a do Matlab aplikace vlastní typ souboru, soubor *MAT*. Pomocí těchto souborů se mohou přenášet data mezi procedurami v rámci jedné Matlab aplikace, mezi uživateli Matlabu a nebo se mohou tyto data zpracovávat i mimo Matlab pomocí funkcí, které poskytuje knihovna Matlab pro C/C++ a Fortran.

4.4.1.1 Datový typ mxArray

Matlab používá pouze jeden typ objektů a to matici. Používá ji pro uložení všech proměnných i objektů. Datový typ *mxArray* zprostředkovává uložení těchto matic do formátu použitelného pro C/C++ a Fortran.

Přehled informací, které jsou uchovávány v *mxArray*:

- typ dat,
- dimenze matice,
- data spojená s maticí,
- jestliže matice obsahuje numerické hodnoty, tak zda jsou reálné nebo komplexní,
- jestliže matice obsahuje struktury nebo objekty, tak počet jejich položek a jejich názvy,
- …

4.4.1.2 Operace se souborem MAT

Knihovna pro práci se soubory *MAT* obsahuje funkce pro otevření a uzavření souboru. Dále umožňuje získat seznam názvů všech matic ze souboru. Pomocí těchto názvů lze pak přistupovat k jednotlivým maticím. Lze číst jak informace o typu matice, tak samotná data a samozřejmě i smazat matici ze souboru. V rámci programu používající tuto knihovnu lze vytvořil vlastní matici a uložit ji do souboru. Nevýhodou je, že knihovna nemá prostředky na zpracování uživatelem definovaných nových objektů.

4.4.1.3 Výměna dat mezi platformami

Soubory *MAT* lze přenášet libovolně mezi platformami. Binární data obsažená v souboru sice nejsou nezávislé na platformě, ale pomocí hlavičky souboru Matlab rozpozná, že se jedná o soubor z jiné platformy a provede potřebné konverze pro novou platformu.

4.4.2 Sdílené knihovny

Matlab umožňuje načíst sdílené knihovny vytvořené v jazyce C/C++. Matlab obsahuje funkce pro načtení knihovny, zjištění dostupných funkcí z knihovny, volání funkcí a odpojení knihovny a uvolnění paměti po ní.

4.4.2.1 Omezení pro sdílené knihovny

1) Sdílené knihovny psané v jazyce C++ se mohou použít pouze v případě, že všechny funkce budou deklarovány jako *extern "C"*.
2) Bitová pole musí být převedena na typ *int* nebo na typ ekvivalentní typu *int*.
3) Při definici výčtového typu nelze použít datový typ *char*. Místo zapsání znaků se musí použít jejich ordinální hodnoty.
4) Ve sdílených knihovnách nesmí být použit datový typ *union*.
5) Ne všechny překladače jsou v Matlabu podporovány. Vždy je nutné používat jen překladače ze seznamu podporovaných překladačů.
6) Není možné používat vnořené struktury nebo struktury obsahující ukazatele na struktury.
7) Rozhraní pro sdílené knihovny nepodporuje ukazatele na funkce a víceúrovňové ukazatele.
8) Nelze použít funkce s proměnným počtem parametrů. K takovýmto funkcím se musí vytvořit alias funkce pro každou kombinaci parametrů a teprve v těchto funkcích lze zavolat původní funkci s proměnným počtem parametrů.

4.4.2.2 Předávání parametrů

Matlab dokáže převést základní skalární datové typy jazyka C/C++ na typy Matlabu a naopak. Dále dokáže převádět ukazatele maximálně do zanoření 2. Veškeré neskalární datové typy musí být v C/C++ funkcích zpracovávané jako předávané odkazem. Někdy může nastat komplikace s předáváním vícedimenzionálních polí. Matlab může změnit dimenze polí. Proto je nutné po získání pole z C/C++ funkce zavolat v Matlabu funkci pro korekci dimenze. Matlab automaticky převádí argumenty na typ očekávaný v externí funkci. Dokonce pokud je předán funkci, která očekává argument předaný odkazem, argument předaný hodnotou, tak Matlab vytvoří ukazatel na tento argument a tento vloží jako správný argument.

4.4.3 Soubory MEX

Matlab umožňuje volat podprogramy psané v jazyce C/C++ a Fortran přímo z Matlabu jako by to byly vestavěné funkce. Soubory MEX obsahují podprogramy, které Matlab dynamicky přilinkuje a jeho interpret je při zavolání vykoná. Soubory MEX jsou v podstatě podobným rozhraní jako sdílené knihovny.

4.4.3.1 Vytvoření souboru MEX

Vytvoření zdrojového souboru MEX spočívá v úpravě obyčejného zdrojového souboru podporovaných jazyků. Tento soubor obsahuje výpočetní rutinu. Po této rutině se musí zapsat tzv. *gateway* rutina, která slouží jako rozhraní mezi výpočetní rutinou a Matlabem. V *gateway* rutině se ověřují vstupní a výstupní parametry funkce, provádí se převod vstupních parametrů na datové typy použitelné ve zdrojovém jazyce a připravují se proměnné na výstupní data. Dále je vhodné používat

makra z knihoven Matlabu pro kompatibilitu mezi platformami (například nahradit datový typ proměnné pro velikost matice datovým typem, který je v knihovně Matlabu pro tento účel vytvořen). Po této úpravě je možné v Matlabu přeložit tento soubor do binárního souboru, který se dynamicky přilinkuje. Název tohoto souboru se musí shodovat s názvem funkce, kterou obsahuje. Matlab při volání funkce prochází složky, ve kterých očekává binární soubory, a hledá shodu se jménem této funkce.

4.4.4 Využití výpočetního jádra Matlabu

Knihovna výpočetního jádra Matlabu obsahuje funkce, které dovolují zavolat Matlab a využít ho jako výpočetní stroj. Matlab mohou využívat programy psané v jazycích C/C++ a Fortran. Tyto programy komunikují s procesem Matlabu pomocí rour na unixových platformách nebo rozhraní *Component Object Model* na platformě Microsoft Windows. Knihovny Matlabu umožňují spustit a ukončit proces Matlabu, posílat data mezi programem a Matlabem a posílat příkazy, které mají být provedeny v Matlabu. Tímto způsobem lze tedy využít Matlab jako výkonnou matematickou knihovnu. Výhodou je, že program nepotřebuje přilinkovat celý program Matlab, ale pouze jeho výpočetní jádro. Využití tohoto rozhraní se dá uplatnit např. jako architektura klient – server, kdy výpočetní jádro Matlabu bude umístěné na serveru a klienti jej mohou přes uživatelské rozhraní využívat.

4.4.5 Použití COM objektů

Microsoft *Component Object Model* (*COM*) poskytuje framework pro integraci binárních softwarových komponent do aplikací. Zdrojový kód těchto komponent může být napsán v libovolném jazyce, který podporuje COM. Výhodou je, že k aplikaci se přidává již zkompilovaná komponenta a není tedy potřeba při aktualizaci komponenty kompilovat celou aplikaci.

COM objekt je komponenta, která v sobě zapouzdřuje data a implementaci. Každý objekt poskytuje své rozhraní, které se skládá z vlastností, metod a událostí. COM klient je aplikace, která využívá COM objekty. COM objektům, která naopak poskytují svou funkcionalitu programům, se říká COM servery. Každý COM objekt je identifikován pomocí programového identifikátoru *ProgID*.

4.4.5.1 COM servery

Rozeznávají se tři druhy serverů:

1. *Automation* servery – servery, které jsou založeny na rozhraní *IDispatch*. Tyto servery mohou používat klienti všech typů.
2. *Custom* servery – servery, které implementují rozhraní *IUnknown*. Jsou určeny pro speciální použití.
3. *Dual* servery – servery, které implementují obě rozhraní.

Existují tři různá nastavení pro server:

1. *In-process* server – komponenta, která je implementována jako dynamická sdílená knihovna nebo jako ovládací prvek *ActiveX*. Tato komponenta běží ve stejném procesu jako klient a sdílí adresový prostor, takže je komunikace mezi nimi velmi rychlá.
2. Lokální *out-of-process* server – komponenta, která je implementována jako spustitelný soubor. Spouští se ve vlastním procesu, ale na stejném počítači.
3. Vzdálený *out-of-process* server – pro tento typ serveru platí to samé, co pro lokální, pouze s tím rozdílem, že komponenta běží na jiné stanici než klient. Proto je komunikace nejpomalejší ze všech nastavení serveru.

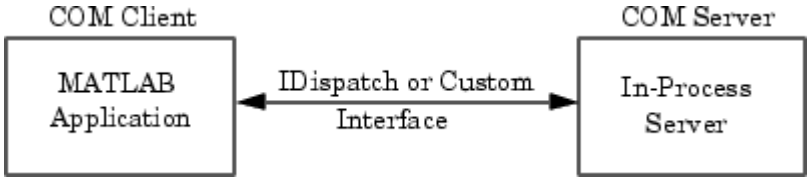
4.4.5.2 COM rozhraní

Existují tři druhy rozhraní:

1. *IUnknown* – toto rozhraní musí povinně implementovat všechny COM objekty. Umožňuje vytvářet nové reference na objekt nebo je rušit a pomocí něj lze získat reference na jiná rozhraní objektu. Všechna ostatní rozhraní jsou odvozena z tohoto.
2. *IDispatch* – toto rozhraní poskytuje informace, které vlastnosti a metody COM objekt poskytuje, a dovoluje spouštět metody a přistupovat k vlastnostem.
3. *Custom* – vlastní rozhraní odvozené od rozhraní *IUnknown*

4.4.5.3 Podporovaná nastavení klient/server v Matlabu

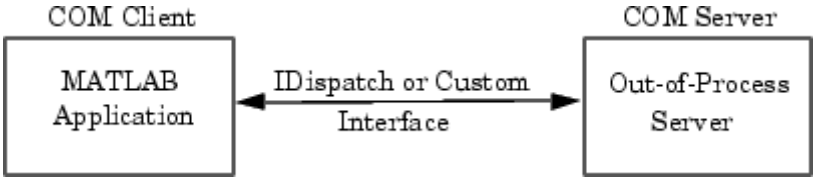
1. Matlab klient a in-process server



4 In-process server

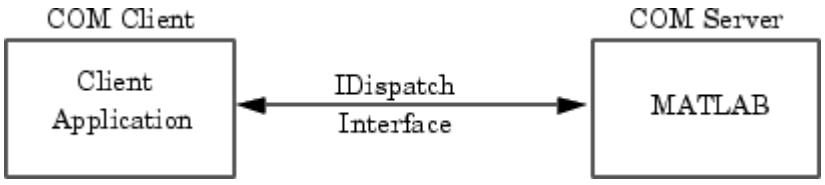
Jako in-process zde může vystupovat buď dynamická sdílená knihovna nebo ovládací prvek ActiveX

2. Matlab klient a out-of-process server



5 Out-of-process server

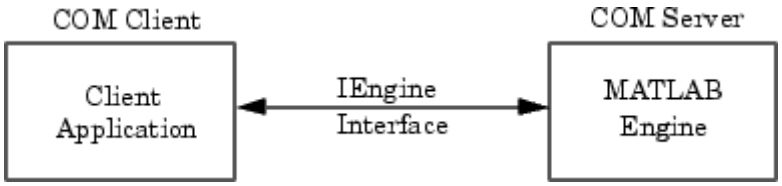
3. Klient a Matlab automation server



6 Automation server

Matlab může být klientskou aplikací spuštěn buď jako sdílený nebo jako dedikovaný server podle toho, jestli běží na vzdáleném nebo lokálním systému.

4. Klient a Matlab engine server



7 Matlab engine

Tento způsob využití výpočetního jádra Matlabu je rozebrán v kapitole 3.4.4.

4.4.5.4 Práce s COM objekty

Vytváření COM objektů

Pro vytváření COM objektů poskytuje Matlab dvě funkce:

1. *actxcontrol* – vytváří instanci ovládacího prvku ActiveX a vrací ukazatel na primární rozhraní objektu. Jako argumenty potřebuje funkce ProgID vytvářeného ovládacího prvku a další hodnoty jako jsou pozice a velikost ovládacího prvku atd.
2. *actxserver* – vytváří server implementovaný knihovnou nebo spustitelným souborem. Stejně jako předchozí funkce i tato vrací ukazatel na primární rozhraní objektu.

Objekty je dále možné ukládat a číst z a do souboru.

Vlastnosti COM objektů

K vlastnostem objektů je možné přistupovat pomocí funkcí *get* a *set* anebo tečkové notace. Tečkovou notaci nelze použít u privátních vlastností a u vlastností s argumenty. Zajímavé je, že názvy vlastností se mohou zkrátit do takové délky, aby stále jednoznačně identifikovali vlastnost v rámci

objektu. Dále je možné objektům přidávat nové vlastnosti a tyto vlastnosti následně mazat. Nelze však mazat původní vlastnosti objektu.

Metody COM objektů

Pro spuštění metody je více způsobů syntaxe:

1. pomocí tečkové notace, tedy *objekt.metoda(arg1,..., argn)*
2. syntaxe Matlabu, tedy *metoda(objekt, arg1,..., argn)*
3. využití funkce *invoke* se syntaxí *invoke(metoda, objekt, arg1, ..., argn)*.

Události COM objektů

Matlab nabízí funkce pro zjištění událostí. Dále je možné pro každou událost zaregistrovat jednu či více funkcí, které se spustí, když událost nastane.

Předchozí text vychází z [11]

5 Návrh řešení

Po prostudování výše uvedených rozhraní pro přístup k simulačním platformám různých jazyků je dalším krokem návrh způsobu využití těchto rozhraní pro kosimulaci se simulační platformou produktu Codasip. Jelikož Codasip simulační platforma je implementována v jazyce C/C++, pro návrh byla vybrána ta rozhraní, která podporují tento jazyk. Jsou to tedy VHDL FLI, Verilog PLI, SystemVerilog DPI a pro Matlab sdílené knihovny.

5.1 Základní koncept návrhu

Návrh začlenění rozhraní do Codasip simulační platformy má podobnou podobu pro všechna vybraná rozhraní. Všechna rozhraní totiž vždy přilinkují Codasip simulační platformu jako sdílenou knihovnu. Vždy je nutné vytvořit inicializační funkci pro ustavení spojení mezi simulačními platformami, funkci pro spuštění Codasip simulační platformy, poté funkce pro synchronizaci a nakonec funkce pro výměnu informací. Další částí pak bude vytvoření zdrojových souborů v cizím jazyce, které budou kosimulační rozhraní využívat. Nyní uvedu návrh jednotlivých funkcí pomocí pseudokódu a poté použití těchto funkcí pro jednotlivá rozhraní.

5.1.1 Inicializační funkce

Inicializace rozhraní bude obsahovat pro téměř všechna rozhraní stejnou funkcionalitu. Prvním krokem bude inicializace synchronizace. Musí se nastavit hodnoty všech proměnných potřebných pro synchronizaci na výchozí hodnotu a také inicializovat zámek pro přístup k těmto proměnným. Pokud rozhraní nabízí možnost reagovat na změny hodnot proměnných v simulaci, tak se jako další krok zaregistrují funkce pro výměnu informací ve směru od cizí simulační platformy k simulační platformě Codasip. Taktéž funkce pro synchronizaci v tomto směru se musí zaregistrovat. Inicializační funkce také musí přijímat jako argument informace potřebné pro spuštění simulační platformy Codasip. Každá simulační platforma předává vstupní parametry jiným způsobem, proto bude nutné vytvořit převod pro každé podporované rozhraní zvlášť. Argumenty se převedou ve všech rozhraních na stejnou řetězcovou reprezentaci. Simulátory Codasipu mají jako povinné vstupní argumenty název projektu, název souboru s přeloženou aplikací a název výstupního souboru pro uložení výstupu simulace. A nakonec se spustí Codasip simulační platformu v novém vlákně. Tím bude jedno vlákno určeno pro komunikaci mezi simulačními platformami a druhé pro běh Codasip simulační platformy.

Zápis funkce pomocí pseudokódu:

```
inicializace ()
{
        inicializace proměnných pro synchronizaci;
        inicializace zámku pro synchronizaci;
        registrace funkcí pro synchronizaci;
        registrace funkcí pro přenos dat;
        převod vstupních argumentů;
        vytvoření nového vlákna pro běh simulační platformy;
}
```

5.1.2 Funkce pro spuštění simulační platformy

Pro spuštění simulační platformy bude vytvořena nová funkce, která převede vstupní argumenty předané z cizí simulační platformy inicializační funkci na argumenty, které vyžaduje Codasip simulační platforma (argumenty *argc* a *argv* běžné hlavní funkce C). Po převodu argumentů se zavolá hlavní funkce simulační platformy s těmito argumenty a tím se spustí celá simulace. Pro všechny možnosti tvorby kosimulačního rozhraní bude tato funkce vždy stejná, nikde k ní nepřibude žádná zvláštní funkčnost, proto už ji nebudu podrobněji rozebírat v návrhu jednotlivých rozhraní.

5.1.3 Funkce pro synchronizaci

Pro synchronizaci mezi simulačními platformami musí být vytvořeny dvě funkce. Jedna pro synchronizaci ze strany Codasip simulační platformy a druhá pro cizí simulační platformu. Na obrázku č. 8 je ukázáno prokládání vykonávání těchto funkcí. Synchronizační funkce musí zajistit, že se bude pravidelně střídat provádění cyklů z jedné a druhé simulační platformy. Čili po ukončení cyklu jedné platformy přijde na řadu druhá. Pokud by se prováděly cykly z obou platforem zároveň, mohlo by docházet k situacím, že si simulační platformy budou přepisovat navzájem data, potřebná pro jejich běh.

V roli hlavní řídící synchronizační funkce bude vystupovat synchronizační funkce na straně cizí platformy. Tato funkce bude mít na starost povolení spuštění cyklu simulace platformy Codasip a čekání na její ukončení. Tím, že tato funkce bude součástí cizí simulační platformy, ať už ve formě volané funkce nebo funkce zastupující proces simulace, tak je zajištěno, že se během jejího provádění nemění hodnoty společných proměnných z této strany simulace. To samé platí i v opačném směru. Jelikož tato funkce bude prakticky spouštět provádění cyklu simulace platformy Codasip a zároveň čekat na její ukončení, tak se nemůže stát, že by se společné proměnné měnily ze strany platformy Codasip při běhu cizí simulace. Další funkcí řídící synchronizační funkce bude kontrola, zda simulace simulační platformy Codasip již neskončila. Tato kontrola zde musí být, aby nedošlo k situaci, při

které by cizí simulace čekala donekonečna na provedení dalšího cyklu simulace Codasipu v situaci, kdy tato simulace již skončila. Jestliže již skončila, tak se synchronizační funkce ukončí bez čekání a pokračuje pak pouze simulace na straně cizí simulační platformy.

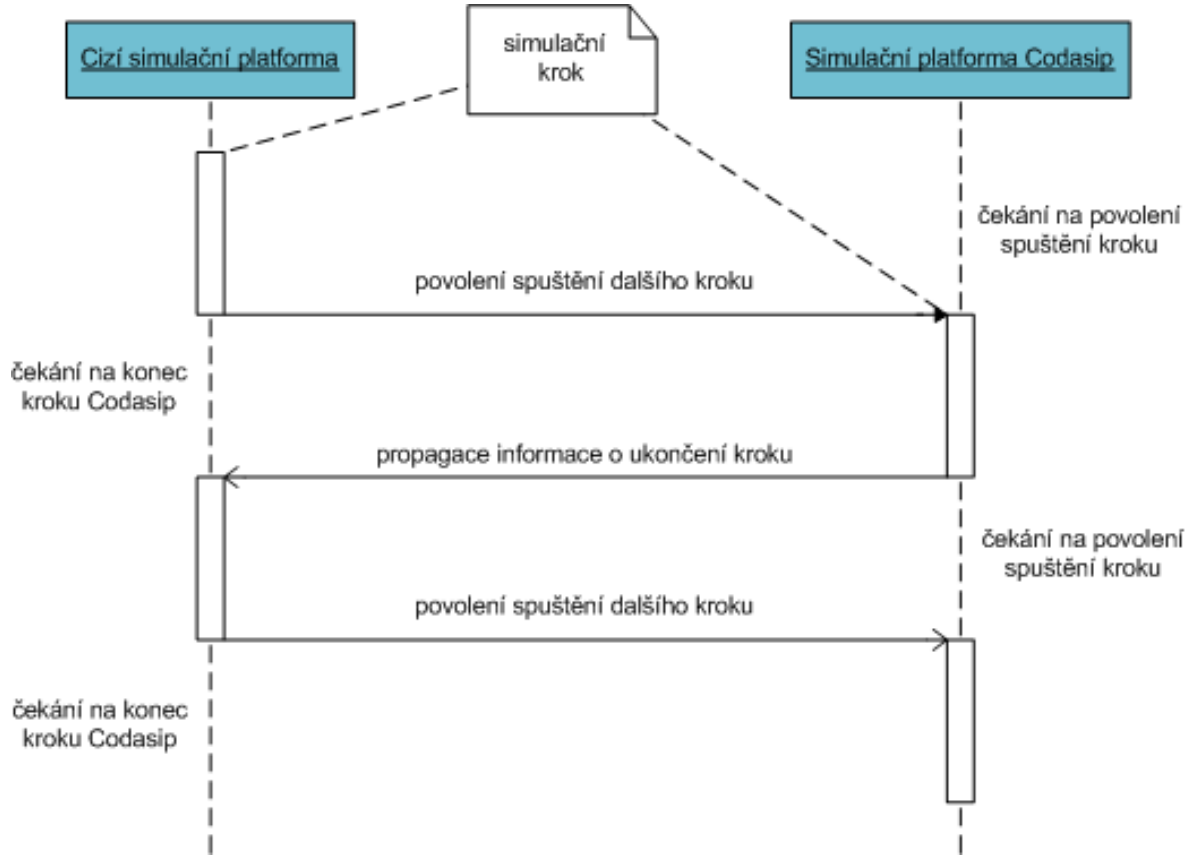
Synchronizační funkce ze strany simulační platformy Codasip bude tedy ve vedlejší řízené roli. Její úlohou bude pouze čekání na povolení zahájení provádění dalšího simulačního cyklu a po jeho provedení zprostředkování informace o ukončení cyklu.

Zápis synchronizační funkce na straně cizí platformy pomocí pseudokódu:

```
synchronizace()
{
    kontrola konce simulace;
    čekání na uvolnění zámku;
    zamčení zámku;
    povolení spuštění dalšího kroku v simulační platformě Codasip;
    uvolnění zámku;
    čekání na ukončení kroku v simulační platformě Codasip;
    čekání na uvolnění zámku;
    zamčení zámku;
    potvrzení ukončení kroku v simulační platformě Codasip;
    uvolnění zámku;
}
```

Zápis synchronizační funkce na straně simulační platformy Codasip pomocí pseudokódu:

```
synchronizace()
{
    čekání na uvolnění zámku;
    zamčení zámku;
    propagace informace o ukončení kroku v simulační platformě Codasip;
    uvolnění zámku;
    čekání na povolení spuštění kroku v simulační platformě Codasip;
}
```



8 Synchronizace

V Codasip simulační platformě bude volání této funkce při generování simulační platformy vloženo na začátek simulačního kroku. Při prvním volání synchronizační funkce simulační platformy Codasip je propagace informace o ukončení jednoho simulačního kroku bezvýznamná a druhá synchronizační funkce na ni nebere ohled. Ta totiž začíná povolením dalšího kroku a pak čeká na dokončení tohoto kroku. Z tohoto vyplývá, že je nezbytně nutné správně nastavit v inicializační funkci synchronizační proměnné, aby nenastal případ, že by se vykonal první krok simulace bez náležitého povolení.

5.1.4 Funkce pro přenos dat

Funkce pro přenos dat budou dvojího typu a to na přenos od Codasip simulační platformy k cizí simulační platformě a naopak. V případě přenosu od Codasip simulační platformy se musí vytvořit funkce, která bude plnit úlohu ovladače vstupních proměnných v cizí simulační platformě (např. signál, port …). Tedy bude měnit v předem definovaných intervalech hodnoty daných proměnných. Simulační platforma Codasip neumožňuje zavést funkci, která by reagovala na změnu hodnoty nějaké její proměnné. Tato funkce se musí tedy volat v předem definovaném čase simulace. Jako vhodnou dobu pro zápis změn ze simulační platfomy Codasip do cizí simulační platformy jsem vybral konec jednoho simulačního kroku Codasipu. Jelikož cizí simulace vždy čeká na dokončení

tohoto kroku a pokračuje ve své činnosti až poté, tak tím bude zajištěno, že pro své výpočty bude mít k dispozici vždy nové hodnoty ze simulace Codasip. Pro přenos v druhém směru se vytvoří funkce v podobě callbacku, který bude vyvolán při změně hodnoty nějaké proměnné v cizí simulační platformě a předá tuto hodnotu Codasip simulační platformě.

Zápis ovládací funkce v pseudokódu:

```
ovladač(){
        pro všechny vstupní proměnné
                získání přístupu k proměnné;
                převod nové hodnoty na cizí datový typ;
                zápis nové hodnoty proměnné;
}
```

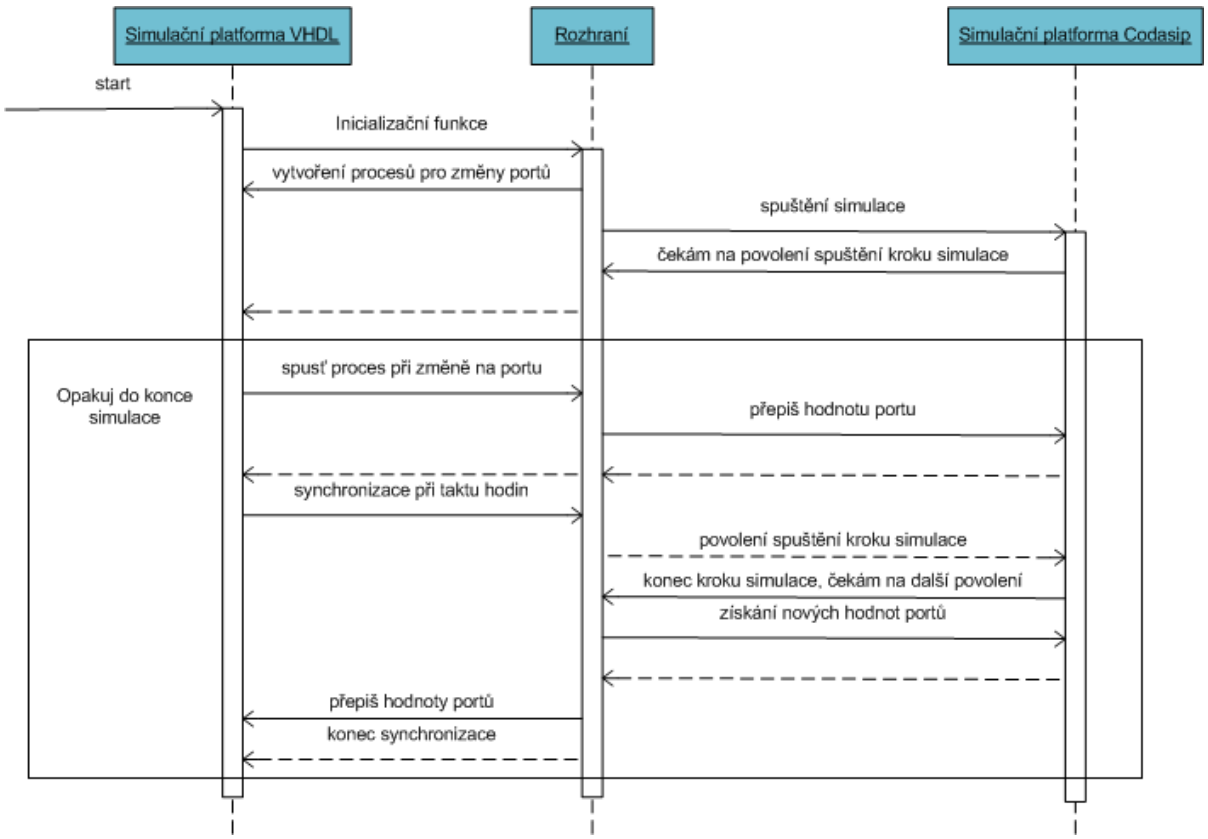
Zápis callbacku v pseudokódu:

```
callback(){
        získání přístupu k proměnné;
        přečtení hodnoty proměnné;
        převod hodnoty na datový typ C;
uložení hodnoty do datových struktur simulační platformy Codasip;
}
```

5.1.5 Zdrojový soubor v cizím jazyce

Zdrojový soubor v cizím jazyce se bude vytvářet hlavně kvůli uživateli. Tím, že se vytvoří tento soubor se zdrojovým kódem, který využívá navržené kosimulační rozhraní, se uživateli usnadní práce s rozhraním. Z názorné ukázky použití nejlépe pochopí používání rozhraní. U každého jazyku bude zdrojový soubor mít jinou podobu, proto uvedu celý návrh těchto souborů pro každý jazyk zvlášť. Jedinou společnou záležitostí ve všech souborech je nutnost předat parametry spuštění simulátoru Codasip. Všechna rozhraní dovolují předávat nějakým způsobem řetězce znaků. Proto bude tato možnost využita u všech rozhraní. Uživatel bude mít vždy přístup k těmto parametrům a může je měnit. Musí mu to být umožněno, protože v parametrech se předává například soubor s přeloženým kódem asembleru, který obsahuje program pro simulaci. Uživatel tedy musí mít možnost simulovat na daném hardwaru více programů a ne jenom jeden výchozí.

5.2 VHDL FLI



9 Diagram průběhu kosimulace VHDL - Codasip

Obrázek č. 9 ukazuje posloupnost volání jednotlivých funkcí kosimulačního rozhraní mezi simulačními platformami VHDL a Codasip.

Inicializační funkce

Inicializační funkce pro FLI bude plnit ty příkazy, které byly popsány výše plus ještě bude registrovat callback funkci pro uvolnění paměti alokované v FLI funkcích při ukončení simulace. Registrace funkcí pro přenos dat a synchronizace bude mít podobu vytvoření nových procesů VHDL. Jeden proces bude sloužit pro synchronizaci a bude reagovat na každý nový takt hodin ve VHDL. Dále pro každý z výstupních signálů z VHDL bude vytvořen jeden proces, který bude reagovat na jeho změny. Posledním úkolem inicializační funkce je převod vstupních argumentů. FLI předává vstupní argumenty inicializační funkci v podobě jednoho řetězce. Přináší to zjednodušení, protože argumenty se tedy nemusí převádět a pouze se v nezměněné podobě předají funkci pro spuštění simulační platformy.

Synchronizační funkce

Jak bylo napsáno výše, synchronizační funkce pro synchronizaci ze strany VHDL bude mít podobu procesu, který bude citlivý na změnu hodinového signálu VHDL. Při každém taktu

hodinového signálu VHDL tedy povolí provedení jednoho kroku v Codasip simulační platformě a po jeho provedení pokračuje znovunastavením spuštění procesu na další změnu hodinového signálu.

Funkce pro přenos dat

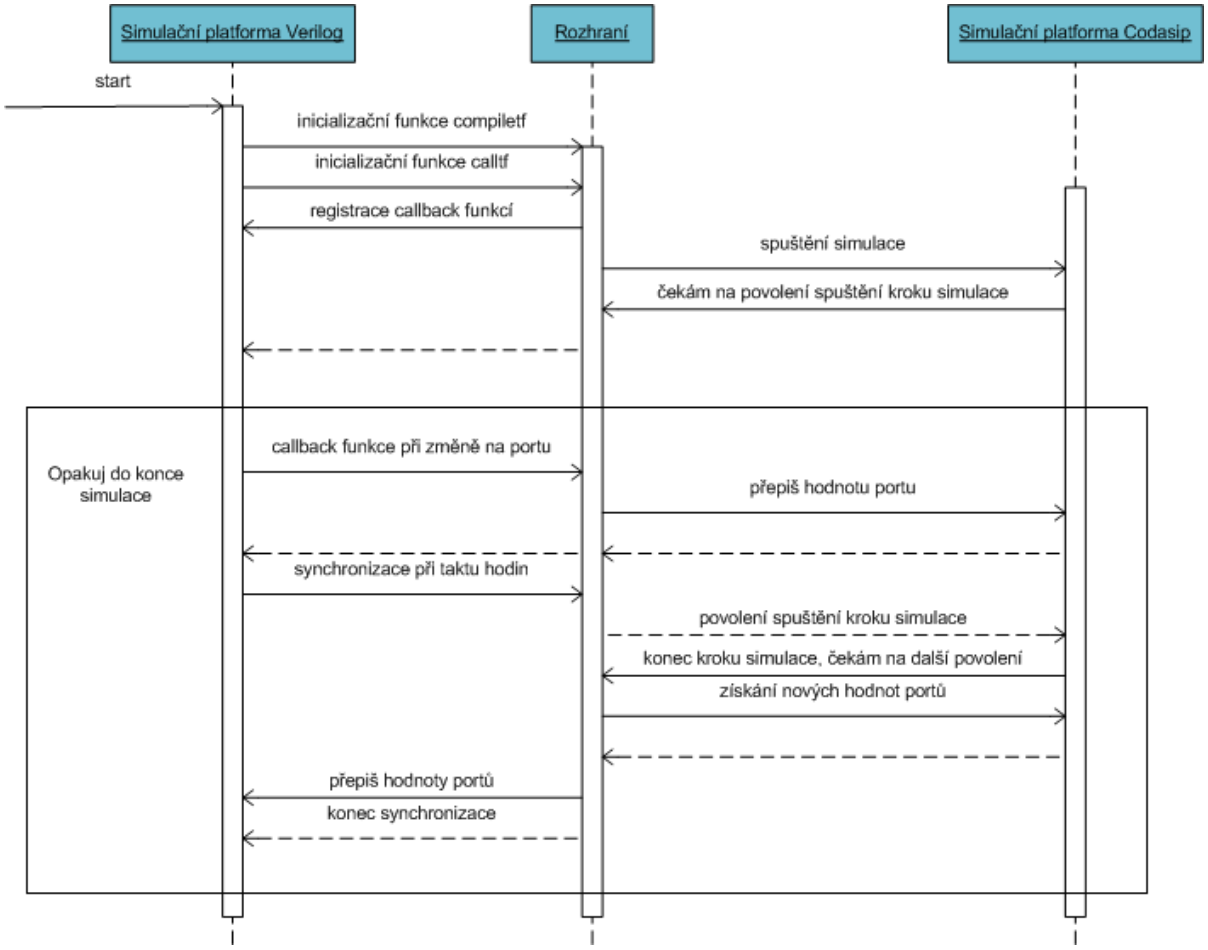
FLI nabízí rutiny jak pro čtení hodnot VHDL proměnných, tak pro zápis do nich. Proto půjdou funkce pro přenos dat implementovat přesně tak, jak je uvedeno v pseudokódu. Pak také rozhraní FLI obsahuje definice datových typů pro reprezentaci hodnot proměnných z VHDL v jazyce C/C++. Při přenosu dat mezi simulačními platformami pak ještě bude nutné převést tyto typy na typy používané v simulační platformě Codasip. Dále lze pomocí FLI nastavit nově vytvořený proces VHDL citlivý na změnu nějakého signálu a k tomuto procesu přiřadit obslužnou funkci. Každému výstupnímu signálu VHDL tedy bude pomocí FLI přiřazen jeden proces s vlastní obslužnou funkcí, která vždy při změně signálu přepíše jeho novou hodnotu na stranu simulační platformy Codasip. Tím je zajištěn přenos dat od VHDL simulační platformy ke Codasip simulační platformě. Opačný směr bude zajišťovat jedna společná funkce pro všechny vstupní signály VHDL.

Zdrojový soubor VHDL

Při vytváření rozhraní mezi simulačními platformami se také automaticky generuje zdrojový kód v jazyce VHDL, který se dále bude používat pro kosimulaci. Uživatel tak nemusí sám hledat, jaké rozhraní poskytuje hardware simulovaný pomocí Codasipu, ale již bude mít k dispozici soubor s vytvořenou entitou, která toto rozhraní popisuje. Tato entita tedy bude pro uživatele představovat jakousi černou skříňku simulovaného hardwaru. Má informaci pouze o rozhraní a od implementace vnitřních funkcí je odstíněn.

Ve zdrojovém souboru VHDL bude entita a jako porty budou použity všechny porty simulovaného hardwaru. Dále se přidají dva jednobitové porty entity, jeden pro hodinový signál a druhý pro signál reset. Jako architektura této entity bude vystupovat cizí architektura, jak byla popsána při analýze rozhraní FLI. Jelikož tato architektura neumožňuje zápis činnosti jinak než pomocí funkcí jazyka C přes rozhraní FLI, tak bude vytvořena další entita, pomocí níž bude uživatel smět ovládat cizí entitu. Tato cizí entita pak bude v nové entitě figurovat jako jedna komponenta. Pomocí signálů nové entity se pak bude dát v její architektuře komponenta ovládat pomocí kódu VHDL. Příklad zdrojového kódu VHDL pro kosimulaci je uveden v Příloze 2.

5.3 Verilog PLI



10 Diagram průběhu kosimulace Verilog – Codasip

Pro kosimulaci se simulační platformou Verilogu bude použit standard PLI 2.0, protože při vydání tohoto standardu bylo doporučeno starší již nepoužívat i když zakázán nebyl. Dalším důvodem je, že tento standard byl tvořen kvůli snadnější práci na implementaci. Nabízí snadnější přístup k vnitřním datovým strukturám simulační platformy Verilog pomocí mnohem méně funkcí než v případě standardu PLI 1.0 při zachování veškeré funkcionality.

Samotné propojení simulačních platforem bude vytvořeno velmi podobně jako v případě FLI. Celá Codasip simulační platforma se připojí k simulační platformě Verilog jako nová uživatelsky definovaná systémová úloha, která se zavolá při inicializaci Verilog modelu. PLI rutiny *compiletf* a *calltf* nově definované úlohy se budou starat o propojení obou simulačních platforem. Posloupnost volání jednotlivých funkcí ukazuje obrázek č. 10.

Inicializační funkce

Jelikož simulační platforma bude spuštěna pomocí zavolání uživatelsky definované systémové úlohy, tak je potřeba předat parametry spuštění simulační platformě pomocí této úlohy. Proto bude vytvořena PLI rutina *compiletf*, která bude mít za úkol ještě před spuštěním simulace

zkontrolovat argumenty volání. Pokud by parametry neodpovídaly, je možné zrušit provádění celé simulace ještě v elaborační části.

Rutina *calltf*, která bude vykonána při volání systémové úlohy, bude plnit funkci inicializační funkce. Stejně jako rozhraní FLI nabízí PLI možnost registrace callback funkcí pro změny proměnných Verilogu. PLI předává rutině *calltf* všechny argumenty, které jsou uvedeny ve volání uživatelsky definované systémové úlohy nebo funkce. Proto inicializační funkce musí převést všechny argumenty na řetězce a předat je spouštěcí funkci Codasip simulační platformě.

Synchronizační funkce

Synchronizační funkce PLI bude mít podobu callback funkce. Ta bude reagovat na změny hodinového signálu a při jeho nástupné hraně povolí další simulační krok simulační platformy Codasip.

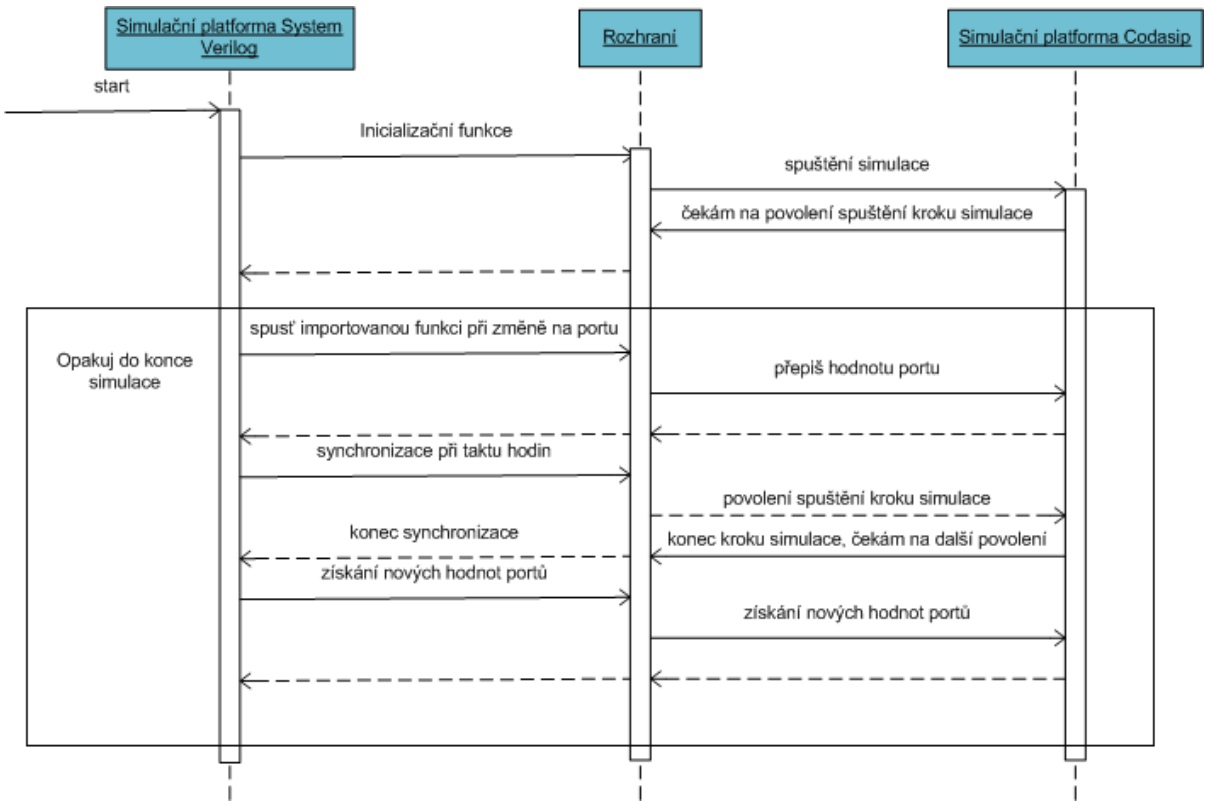
Funkce pro přenos dat

Funkce pro přenos dat budou vytvořeny stejně jako v případě FLI. Rozdíl je pouze v tom, že není potřeba vytvářet nové procesy. Stačí zaregistrovat funkce pro každý výstupní a vstupně-výstupní signál Verilogu, který chceme přepsat na stranu simulace Codasipu. Stejně jako u FLI lze funkci vyvolat jako reakci na změnu hlídaného signálu. Vstupní a vstupně-výstupní signály Verilogu se pak budou obnovovat pomocí jedné funkce na konci simulačního kroku simulační platformy Codasip. Součástí funkcí pro přenos bude také konverze datových typů, protože rozhraní PLI obsahuje knihovnu s definicí vlastních typů pro hodnoty proměnných Verilogu a bude potřeba tyto typy převést na typy používané simulační platformou Codasip.

Zdrojový soubor Verilog

Jako bude u rozhraní FLI reprezentován simulovaný hardware pomocí entity, tak v případě PLI se bude jednat o reprezentaci pomocí modulu. Modul bude mít stejné porty jako u FLI, tedy jednobitové vstupní porty pro hodinový signál a signál reset a dále všechny porty simulovaného hardwaru. I ve Verilogu se pak určí směr proudu dat portů. V popisu funkcionality modulu se pak objeví v inicializačním bloku volání nově vytvořené uživatelské systémové úlohy. Tím se modul propojí pomocí PLI se simulací hardwaru v Codasipu. K tomuto modulu se pak vytvoří ještě jeden, který bude obsahovat modul s rozhraním jako komponentu. V tomto modulu se vytvoří proměnné pro přístup k jednotlivým portům a pomocí nich se pak bude simulovaný hardware ovládat. Příklad zdrojového kódu Verilogu pro kosimulaci je uveden v Příloze 3.

5.4 SystemVerilog DPI



11 Diagram průběhu kosimulace SystemVerilog – Codasip

Spojení simulačních platforem pomocí DPI je komplikovanější. Pomocí importovaných funkcí totiž nelze přistupovat k simulačnímu času a nelze přiřadit žádnou funkci jako reakci na změnu signálu. Posloupnost volání funkcí rozhraní je ukázána na obrázku č. 11.

Inicializační funkce

Rozhraní DPI nenabízí kromě volání externích funkcí žádnou možnost, jak reagovat na změny signálů. Proto bude úkolem inicializační funkce pouze nastavení hodnot synchronizačních proměnných. Taktéž argumenty je možné předávat pomocí standardních řetězců jazyka C, takže převod argumentů na jiné typy se také nemusí řešit.

Synchronizační funkce

Synchronizační funkce bude mít podobu importované funkce, která se v SystemVerilogu zavolá v bloku *always*, který bude asociován s nástupnou hranou hodinového signálu. Tento blok se tedy provede vždy při taktu hodin a synchronizuje tak obě simulační platformy.

Funkce pro přenos dat

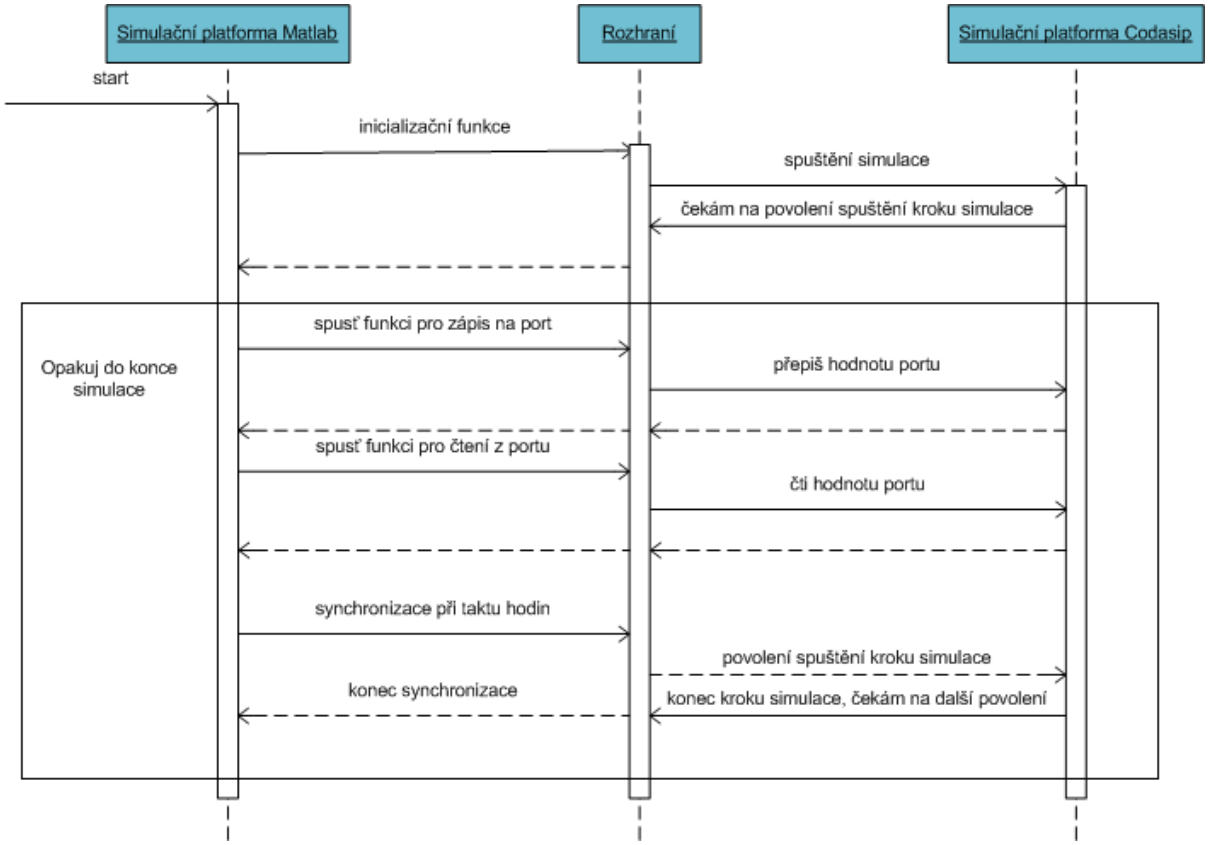
Funkce pro přenos budou pracovat odlišně než v případě FLI a PLI. Funkce pro získávání hodnot výstupních a vstupně-výstupních signálů ze SystemVerilogu bude volána stejně jako synchronizační funkce, tedy z bloku *always*, který bude asociován s proměnnou, jejíž změny chceme odchytávat. Pro každý z těchto signál bude tedy vytvořena samostatná funkce, která bude mít jako

parametr daný signál, jehož hodnota se právě změnila. Pro opačný směr, tedy řízení hodnot vstupních a vstupně-výstupních signálů v SystemVerilogu, bude implementována jedna importovaná funkce pro všechny signály. Jako parametry této importované funkce budou sloužit všechny vstupní a vstupně-výstupní signály SystemVerilogu. Rozhraní DPI pak také obsahuje knihovnu s definovanými typy pro reprezentaci proměnných jazyka SystemVerilog v jazyce C. Proto bude ještě zapotřebí konvertovat hodnoty proměnných na typy, se kterými pracuje simulační platforma Codasip.

Zdrojový soubor SystemVerilog

Zdrojový soubor pro rozhraní SystemVerilogu DPI bude obsahovat vše, co soubor pro rozhraní Verilogu PLI. Oproti Verilogu pouze přibudou volání všech importovaných funkcí. Tedy v bloku *always*, který bude reagovat na vzestupnou hranu hodinového signálu, bude volána funkce pro synchronizaci a následně po ní funkce pro přenos dat ve směru od simulační platformy Codasip k platformě SystemVerilog. V blocích *always*, které budou reagovat na změny jednotlivých výstupních a vstupně-výstupních signálů ze SystemVerilogu, se budou vyvolávat funkce pro přenos hodnoty příslušných signálů v opačném směru. Příklad zdrojového kódu SystemVerilogu pro kosimulaci je uveden v Příloze 4.

5.5 Matlab



12 Diagram průběhu kosimulace Matlab – Codasip

Pro spojení Codasip simulační platformy s Matlabem bylo vybráno rozhraní Matlabu pro sdílené knihovny. Matlab může být využíván např. jako generátor analogových signálů. Posloupnost volání funkcí rozhraní je ukázána na obrázku č. 12.

5.5.1 Sdílené knihovny

Rozhraní pro sdílené knihovny nenabízí jiné možnosti, než volání funkcí knihovny stejně jako v případě DPI. Rozhraní pro Matlab bude tedy velmi podobné jako pro SystemVerilog.

Inicializační funkce

Inicializační funkce tohoto rozhraní bude mít za úkol pouze nastavit hodnoty proměnných pro synchronizaci. Nelze vytvářet žádné callback funkce jak pro synchronizaci, tak pro přenos dat. Také předání argumentů simulační platformě Codasip se v tomto případě nemusí řešit, jelikož Matlab dovoluje předání argumentů formou řetězce.

Synchronizační funkce

Toto rozhraní umožňuje pouze volání funkcí ze sdílené knihovny, proto vyvolání synchronizační funkce musí být provedeno z Matlabu a to vždy, když bude k dispozici výsledek pro

nový simulační čas. Celý proces simulace tak je naprosto v režii uživatele, který může spouštět další cyklus simulace podle svého uvážení. Synchronizační funkce tedy bude pouze spouštět jeden cyklus simulace, jiná funkcionalita zde není zapotřebí. Pro účely synchronizace se pro toto rozhraní vytvoří ještě jedna funkce, která bude testovat konec simulace. Opět nejde nijak zařídit, aby se automaticky při ukončení simulace simulační platformy Codasip propagovala informace o konci na stranu Matlabu. Proto bude povinností uživatele při kosimulaci na straně Matlabu vyvolat tuto funkci po každém provedeném simulačním kroku a patřičně reagovat na ukončení simulace.

Funkce pro přenos dat

Stejně jako u synchronizační funkce i zde bude předávání dat mezi simulačními platformami pouze na uživateli. V knihovně budou vytvořeny dvě funkce. Jedna pro zápis hodnoty do některého ze vstupních nebo vstupně-výstupních portů simulovaného hardwaru na straně Codasipu a druhá pro čtení z výstupních nebo vstupně-výstupních portů. Jelikož umístění a parametry volání těchto funkcí bude mít pod správou uživatel a nebudou vyvolávány automaticky, bude nutné do těchto funkcí zabudovat kontroly pro správné volání těchto funkcí. Kontrola bude spočívat v ověření, zda se čte pouze z výstupních a vstupně-výstupních portů a naopak, že se zapisuje do vstupních a vstupně-výstupních portů.

Zdrojový soubor Matlab

Soubory pro VHDL, Verilog a SystemVerilog měly všechny velmi podobnou strukturu. V případě Matlabu však půjde o naprosto rozdílný přístup k tvorbě zdrojového souboru pro kosimulaci. V Matlabu se nebude vyskytovat žádná struktura popisující simulovaný hardware z Codasipu. Skript Matlabu nejprve načte sdílenou knihovnu se simulátorem hardwaru z Codasipu. Po načtení zavolá inicializační funkci. Potom bude následovat cyklus, jehož podmínkou ukončení bude konec simulace na straně Codasipu. Pro podmínku se tedy využije funkce rozhraní simulace testující tuto skutečnost. V každém cyklu se pak jedenkrát zavolá synchronizační funkce, ve které se vykoná jeden krok simulace na straně Codasipu. To je vše, co bude generovaný zdrojový soubor Matlabu obsahovat. Vše ostatní už je na uživateli. V cyklu se mohou volat jakékoli funkce pro zápis na porty simulovaného hardwaru nebo čtení z nich. Příklad zdrojového kódu Matlabu pro kosimulaci je uveden v Příloze 5.

6 Implementace návrhu řešení

Z vytvořených návrhů řešení jsem implementoval kosimulaci simulační platformy Codasip se simulační platformou VHDL, Verilog, SystemVerilog a Matlab. V této kapitole popíši způsob implementace rozhraní mezi těmito platformami. Synchronizační funkce a funkce pro spuštění simulace jsou ve všech případech téměř shodné a proto popis jejich implementace uvedu nezávisle na rozhraních. U jednotlivých rozhraní pak pouze uvedu rozdíly oproti základnímu řešení.

Ve funkcích pro přenos dat mezi simulačními platformami se vždy musí převádět datové typy hodnot proměnných. Simulační platforma Codasip má pro hodnoty portů vlastní datový typ *codasip_int*. Tento typ může reprezentovat porty až do bitové šířky 1024 bitů. Zároveň jsou pro tento typ definovány funkce pro převod na řetězec reprezentující hodnotu proměnné a to v různých číselných základech. V implementaci rozhraní bude výhodná reprezentace hodnoty při dvojkovém základu. Řetězec má vždy plnou délku reprezentace 1024 bitů, tedy 1024 znaků. Při získání řetězce se tedy musí pokaždé vybrat pouze tolik nejnižších bitů, kolik jich má daný port. Stejně tak jsou definovány funkce, které z řetězce vytvoří proměnnou typu *codasip_int*. Pro zápis do portu na straně Codasipu slouží funkce *port_write*, která má jako parametry název portu a hodnotu datového typu *codasip_int*. Pro čtení pak slouží funkce *port_read*, která má jako parametr název portu, jehož hodnotu chceme získat, a hodnota datového typu *codasip_int* je získána jako návratová hodnota funkce.

6.1 Generování funkcí rozhraní

V implementační části mé diplomové práce je kladen největší důraz na optimálnost. Je nutné, aby rozhraní nespotřebovávalo zbytečně velké množství procesorového času. Zdrojové soubory pro vytvoření konkrétního simulátoru Codasipu jsou generovány z popisu konkrétního hardwaru v jazyce CodAL. Proto je možné provádět optimalizace již při sestavení simulátoru na základě známého popisu rozhraní hardwaru. Z toho důvodu se zdrojové soubory rozhraní mezi simulačními platformami generují také speciálně pro každý simulovaný hardware.

Při generování zdrojových souborů se popis hardwaru v jazyce Codal převádí na datové struktury obsahující popis jednotlivých částí hardwaru. Pro potřebu vytvoření zdrojových souborů rozhraní jsou nezajímavé veškeré struktury popisující funkční části hardwaru. Jediné, co je potřeba, jsou datové struktury obsahující informaci o rozhraní hardwaru. Toto rozhraní je popsáno v datové struktuře *CPort*, která uchovává informace o všech datových portech hardwaru. Pro účel kosimulace jsou zapotřebí názvy portů, jejich bitové šířky a informaci o tom, jestli jsou porty pouze vstupní nebo výstupní nebo vstupně-výstupní. Těchto informací je možné využít při generování zdrojových

souborů, takže se nemusí tvořit funkce pro obecné řešení, které by muselo brát v úvahu libovolný počet portů libovolného typu a bitové šířky.

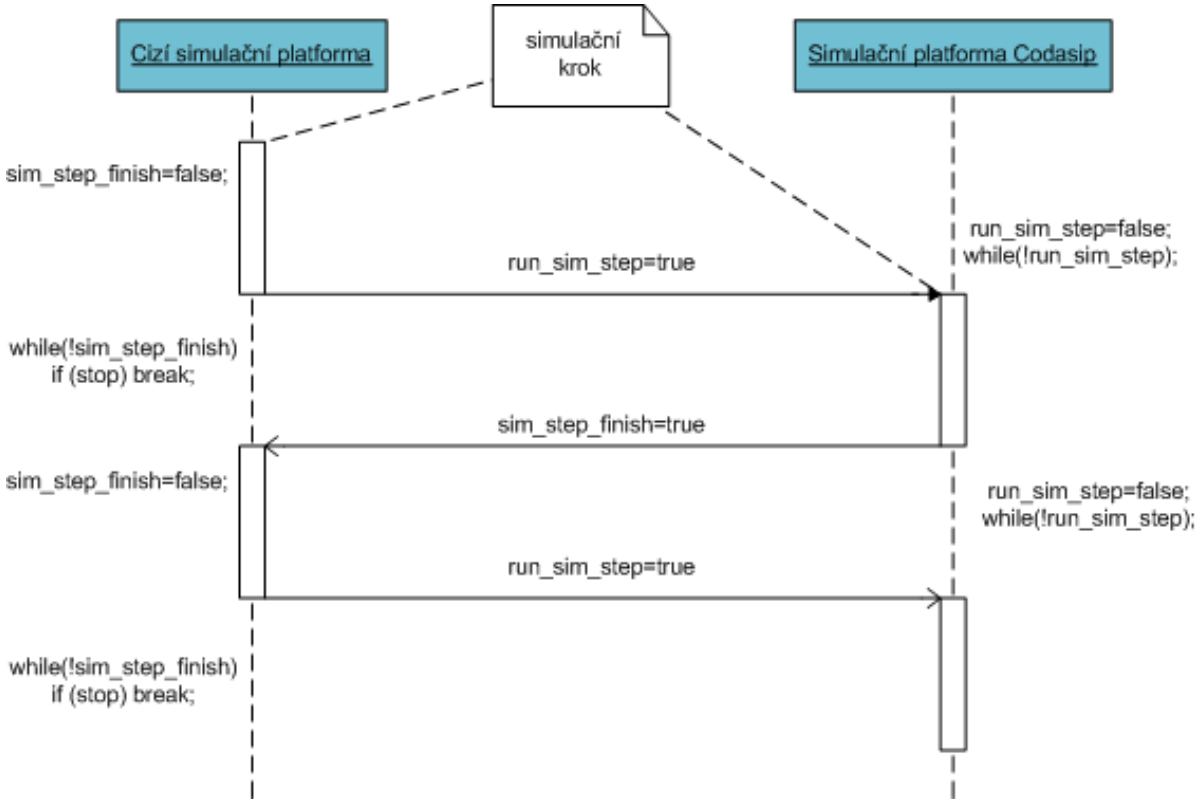
6.2 Synchronizační funkce

Synchronizační funkce jsou základem celé implementace. Od nich se odvíjí celá kosimulace. Proto zde uvedu i samotný zdrojový kód těchto funkcí s vysvětlením jejich činnosti a diagram sekvence na obrázku č. 13, který osvětluje posloupnost kroků při synchronizaci.

Zápis kódu obou synchronizačních funkcí:

```
synchronize_foreign()
{
        if(!stop)
        {
                while(lock);
                lock = true;
                run_sim_step = true;
                lock = false;
                while(!sim_step_finish)
                        if (stop) break;
                while(lock);
                lock = true;
                sim_step_finish = false;
                lock = false;
        }
}
```

```
synchronize_codasip()
{
                while(lock);
                lock = true;
                sim_step_finish = true;
                run_sim_step = false;
                lock = false;
                while(!run_sim_step);
}
```



13 Posloupnost kroků při synchronizaci

Funkce synchronize_foreign je funkce, která se stará o synchronizaci ze strany cizí simulační platformy a funkce synchronize_codasip ze strany simulační platformy Codasip. Jak je patrné, tak obě funkce využívají pouze booleovské proměnné. Synchronizační funkce se provádí každá v jiném vlákně, proto všechny proměnné jsou globální, aby k nim měli přístup obě funkce.

Celá simulace začíná inicializační funkcí, která spustí simulaci Codasipu. Na začátku této simulace se hned zavolá funkce synchronize_codasip. Na začátku nastaví proměnnou sim_step_finish na hodnotu true, čímž se ukončuje jeden krok simulace Codasipu. Při prvním volání této funkce je toto bezvýznamné, protože žádný krok ještě nebyl proveden, ale z hlediska synchronizace to ničemu nevadí. Dalším příkazem se nastaví proměnná run_sim_step na hodnotu false. Tímto je zaručeno, že funkce bude čekat na následující podmínce na povolení spuštění dalšího kroku od druhé funkce.

Druhá synchronizační funkce synchronize_foreign hned na začátku zkontroluje, zda náhodou není konec simulace pomocí proměnné stop. Tato proměnná je vždy na začátku simulace nastavena na false. Simulace Codasipu se odehrává v jedné řídící funkci. A právě hned po dokončení této funkce se přiřadí do proměnné stop hodnota true. Po provedení posledního kroku simulace Codasipu však už nedojde k dalšímu zavolání její synchronizační funkce a nenastaví se tedy proměnná indikující konec jednoho kroku. Místo toho se nastaví právě hodnota proměnné stop. Proto je v každém cyklu kontroly proměnné sim_step_finish ještě kontrolována i proměnná stop, aby na konci simulace nedošlo k uváznutí v tomto cyklu.

Poslední důležitou součástí synchronizačních funkcí je zámek, který je reprezentován proměnnou lock. Synchronizační funkce se pravidelně střídají ve své činnosti a ve chvíli, kdy jedna čeká na změnu sdílené proměnné, druhá do ní zapisuje a naopak. Nemůže se tedy stát, že by obě funkce v jednu chvíli měnily sdílenou proměnnou. Nebezpečí se ale ukrývá ve funkci synchronize_codasip ve dvou navazujících příkazech přiřazení. Kdyby ve funkcích nebyl zámek a provedl by se první přiřazovací příkaz, čímž se dá najevo konec jednoho simulačního kroku, a poté by se pozastavilo provádění vlákna Codasipu, mohlo by dojít k situaci, že by prvně provedlo povolení dalšího simulačního kroku v cizí synchronizační funkci. Při dalším pokračování synchronizační funkce Codasipu by se proměnná run_sim_step nastavila na hodnotu false a tím by došlo k uváznutí obou funkcí. Zámek tedy zajišťuje, že obě přiřazení proběhnou z hlediska synchronizace hned za sebou.

6.3 Funkce pro spuštění simulátoru

Jako vstupní argument přijímá funkce pro spuštění simulátoru jeden řetězec, ve kterém budou všechny parametry spuštění simulátoru Codasip. Tento řetězec se ve funkci rozdělí na jednotlivé parametry a pak se zavolá hlavní funkci simulátoru Codasip a předají se jí tyto parametry. Dělení řetězce je provedeno jednoduchým hledáním mezer v řetězci. Podle těchto mezer se pak řetězec dělí.

6.4 Kosimulace simulačních platforem Codasip a VHDL

Pro implementaci rozhraní mezi těmito simulačními platformami jsem využil výše popsané rozhraní FLI. Jelikož toto rozhraní nabízí možnost přistupovat k vnitřním strukturám simulační platformy VHDL, snažil jsem se tohoto faktu co nejvíce využít při implementaci, aby byl uživatel odstíněn od pro něj vcelku nepotřebnými detaily způsobu fungování rozhraní mezi platformami. Celá kosimulace pak probíhá autonomně bez nutnosti zásahů uživatele.

6.4.1 Zdrojový soubor VHDL

Jak bylo popsáno už v návrhu, ve zdrojovém souboru VHDL bude entita zastupující simulovaný hardware ze strany Codasipu. Jména portů zastupující skutečné porty simulovaného hardwaru pak zůstanou stejná jak na straně VHDL, tak na straně Codasipu. Jednak se tak usnadní práce uživatele s modelem, a také se velmi zjednoduší tvorba rozhraní mezi portem ve VHDL a portem v Codasipu. Jako datový typ portů jsem zvolil *std_logic* pro jednobitové porty a *std_logic_vector* pro vícebitové. Tento typ nejvíce odpovídá skutečnému popisu portů. U portů entity VHDL je dále zapotřebí určit směr toku dat. Pro datové porty simulovaného hardwaru se informace o

směru toku dat získají z datové struktury *CPort*. Port pro hodinový signál a signál reset je vždy vstupní.

K takto vytvořené entitě se dále vygeneruje příslušná cizí architektura. V ní se definuje název inicializační funkce, název sdílené knihovny a parametry spuštění. Název inicializační funkce je vždy stejný. Název sdílené knihovny a parametry spuštění se dají zjistit již v době generování souboru a jsou tedy také automaticky doplněny. Tím je uzavřen popis entity zastupující simulovaný hardware.

Druhá entita, která slouží jako ovládání simulovaného hardwaru, bude obsahovat definici komponenty, kterou bude tvořit první entita. Dále pro každý port komponenty bude vytvořen jeden signál. Typ bude stejný jako u portů, tedy *std_logic* pro jednobitové porty a *std_logic_vector* pro vícebitové. V této entitě bude také implementováno řízení hodinového signálu. Proto se v definici entity ještě objeví definice konstanty určující periodu hodinového signálu. V architektuře této entity se pak pomocí VHDL funkce *port_map* propojí definované signály s porty komponenty. Použil jsem bezpečnější způsob zadání napojení signálů na porty, tedy dvojici název signálu – název portu. Dal by se využít i druhý zápis, kde pouze pořadí signálů určuje mapování na porty, ale toto řešení by mohlo vést následně k chybám, pokud by uživatel z nějakého důvodu změnit pořadí portů nebo by definoval nové. Nakonec se v souboru objeví dva procesy pro samotné řízení simulace a hodinového signálu. Proces pro řízení hodinového signálu bude pouze měnit po půlperiodě hodinový signál z 0 na 1 a naopak. Druhý proces v úloze hlavního procesu celé simulace pak bude určovat, kolik taktů se vykoná při běhu simulace.

6.4.2 Inicializace rozhraní

Inicializace rozhraní probíhá ve speciální funkci volané v elaborační části simulace. Inicializační funkce má celkem čtyři parametry, ale z hlediska tvorby rozhraní pro kosimulaci jsou potřebné pouze dva. Prvním je řetězcový parametr, ve kterém jsou předávány parametry pro spuštění simulátoru Codasip. Tento řetězec je v nezměněné podobně předán funkci pro spuštění simulace při vytvoření nového vlákna. Pro vytvoření vlákna jsem použil knihovnu *pthread*, tedy při vytvoření nového vlákna se jako vstupní funkce vlákna vloží funkce pro spuštění simulace a jako parametr funkce právě získaný řetězec.

Druhým parametrem, který je pro tvorbu rozhraní zapotřebí, je seznam portů entity. Z tohoto seznamu se vybrat pomocí FLI funkce *mti_FindPort* všechny identifikátory portů simulovaného hardwaru a identifikátory pro hodinový signál a signál reset. Pokud je port vícebitový, tak ho rozhraní FLI reprezentuje jako signál skládající se z jednobitových elementárních signálů. Proto je nutné pro tyto vícebitové porty získat ještě seznam identifikátorů jejich elementů. Všechny identifikátory se musí uložit do globálních proměnných, protože se k nim bude přistupovat během simulace z více funkcí. Pro hodinový signál, signál reset a pro všechny vstupní a vstupně-výstupní porty

simulovaného hardwaru se vytvoří nové procesy VHDL, každý obsluhovaný vlastní funkcí s citlivostí nastavený na změnu daného portu.

6.4.3 Synchronizační funkce

Synchronizační funkce ze strany VHDL musí být ještě mírně rozšířená oproti její implementaci popsané výše. Při kosimulaci bude tato funkce volána při každé změně hodinového signálu, tedy jak při nástupné, tak při sestupné hraně. Aby se tedy neprovedly dva takty simulace Codasip, zatímco ve VHDL pouze jeden takt, tak na začátku synchronizační funkce musí být podmínka, že synchronizace se spustí pouze, když hodinový signál bude mít hodnotu 1. Po dokončení kroku simulace Codasipu se následně zavolá funkce pro přenos změn výstupních a vstupně-výstupních portů simulovaného hardwaru.

Protože nastavení citlivosti funguje pouze na jednu změnu signálu, tak další přidanou funkčností této funkce je znovunastavení citlivosti procesu na změnu hodinového signálu.

6.4.4 Funkce pro přenos dat

Pro každý vstupní a vstupně-výstupní port hardwaru simulovaného v Codasipu bude vytvořena speciální funkce. Ta na základě identifikace portu na straně VHDL zjistí jeho hodnotu.

VHDL proměnná typu *std_logic*, kterou se reprezentují jednotlivé bity portů ve VHDL, může nabývat následujících hodnot:

- `'U'`: neinicializováno – do proměnné zatím nebylo nic přiřazeno,
- `'X'`: neznámá hodnot – není možné zjistit hodnotu,
- `'0'`: logická hodnota 0,
- `'1'`: logická hodnota 1,
- `'Z'`: stav vysoké impedance,
- `'W'`: slabý signál – není možné určit, zda je hodnota 1 nebo 0,
- `'L'`: slabý signál – pravděpodobně se jedná o logickou hodnotu 0,
- `'H'`: slabý signál – pravděpodobně se jedná o logickou hodnotu 1,
- `'-'`: bezvýznamné – není potřeba znát logickou hodnotu.

Rozhraní FLI mapuje hodnoty tohoto typu na celočíselné hodnoty, kde hodnota odpovídá pořadí v předchozím výčtu. Datový typ *codasip_int* však dokáže popsat jen hodnoty bitů pomocí logických hodnot 0 a 1 (stejně jako datový typ int jazyka C). Proto jsem zavedl mapování *std_logic* na *codasip_int* a to tím způsobem, že logická hodnota 1 se mapuje na číslo 1 a cokoliv jiného než logická hodnota 1 je považováno za logickou hodnotu 0. Při získávání hodnoty portu se pak čtou

jednotlivé bity portu a tvoří se z nich řetězec znaků 0 a 1. Nakonec se tento řetězec převede na datový typ *codasip_int* a hodnota se zapíše do portu na straně Codasipu.

Pro všechny výstupní a vstupně-výstupní porty hardwaru simulovaného v Codasipu pak bude přenos hodnoty portů probíhat v jedné funkci. Pro každý port se přečte hodnota na straně Codasipu, převede se na řetězec a pak se postupně prochází znak po znaku a do jednotlivých bitů portu na straně VHDL se zapisují příslušné hodnoty.

6.5 Kosimulace simulačních platforem Codasip a Verilog

Stejně jako rozhraní FLI i rozhraní PLI pro Verilog umožňuje přístup k vnitřním datovým strukturám simulátoru a proto může rozhraní kosimulace být implementováno skrytě pro uživatele.

6.5.1 Zdrojový soubor Verilogu

Modul sloužící pro kosimulaci bude mít stejné rozhraní jako simulovaný hardware s přidanými porty pro hodinový signál a signál reset. Názvy portů modulu budou stejné jako názvy portů simulovaného hardwaru. I v jazyce Verilog se u portů specifikuje směr toku dat porty, mohou být typu *input*, *output* a *inout*. Směry toku dat portů budou zachovány tak, jak jsou v simulovaném hardwaru. Jako datový typ portů jsem zvolil typ *wire*. S těmito porty se v definici objeví už pouze blok *initial* s voláním uživatelsky definované úlohy.

Druhý modul bude obsahovat definici proměnných pro propojení s předchozím modulem. Tyto proměnné budou opět datového typu *wire*. Propojení s předchozím modulem se provede podobně jako u VHDL pomocí pojmenovaného přiřazení proměnných k portům. V tomto modulu se rovněž připraví generování hodinového signálu. Poslouží k tomu blok *always*, který nebude asociován s žádnou proměnnou. V tomto bloku se po daném časovém intervalu změní hodnota hodinového signálu z 0 na 1 a naopak.

6.5.2 Inicializační funkce

Pro rozhraní PLI se inicializační funkce skládá z dvou částí. V první části se pomocí funkce *compiletf* rozhraní PLI zkontrolují parametry. Nově definovaná uživatelská funkce, která zastupuje rozhraní pro kosimulaci, má dva parametry. Prvním je název Verilog modulu, který je definován ve výše popsaném zdrojovém souboru jako modul pro popis rozhraní simulovaného hardwaru. Druhým parametrem je pak řetězec s parametry spuštění simulátoru Codasip. Ve funkci *compiletf* se nejprve získá identifikace instance uživatelské systémové úlohy. Dalším krokem je potom získání argumentů volání. K tomu slouží funkce rozhraní PLI *vpi_iterate*. Tato funkce bere jako parametry typ dat, které je potřeba získat, a identifikace objektu, v rámci kterého se budou tato data hledat. Pro zjištění

argumentů se tedy zavolá tato funkce s parametry *vpiArgument*, což je předdefinovaná konstanta pro identifikaci argumentů, a identifikací instance systémové úlohy. Tímto se získá seznam argumentů, přes který je možno iterovat pomocí funkcí rozhraní PLI. Zkontroluje se tedy, zda systémová úloha má právě dva parametry a zda první parametr, je identifikace existujícího modulu v kódu Verilogu. Pokud toto není splněno, tak se celá simulace zastaví.

Druhou částí je funkce *calltf*, která se vykoná při zavolání systémové funkce. Tato funkce už odpovídá přesně návrhu inicializační funkce popsaného výše. Stejným způsobem jako ve funkci *compiletf* se získá seznam argumentů. Ze jména modulu se získá jeho identifikace a druhý parametr se předá při vytvoření nového vlákna funkci pro spuštění simulace. Dále se pomocí funkce *vpi_iterate* získají identifikace na všechny porty. Tyto identifikátory budou zapotřebí během celé simulace, proto se musí uložit tak, aby k nim měli přístup všechny callback funkce rozhraní. Při používání rozhraní PLI je doporučeno nevyužívat k tomu účelu globální proměnné, ale raději využít paměťový prostor pro uživatelská data, který umožňuje uložení ukazatele na jakákoliv data. K těmto datům je možné přistupovat ze všech funkcí příslušejících k jedné systémové úloze. Pro uložení identifikátoru se vytvoří struktura obsahující jako položky všechny identifikátory. Do uživatelského paměťového prostoru se pak uloží ukazatel na tuto strukturu. Posledním krokem je pak zaregistrování callback funkcí pro všechny vstupní a vstupně-výstupní porty simulovaného hardwaru. K zaregistrování callback funkce slouží struktura *s_cb_data*. Do této sktruktury se vyplní informace o důvodu vyvolání callback funkce o názvu této funkce, dále pak identifikátor portu, na jehož změnu má funkce reagovat a formát přijetí nové hodnoty portu. Pro formát přijetí jsem zvolil řetězcovou reprezentaci hodnoty portu v binárním kódu kvůli převodu na typ *codasip_int*. Pomocí funkce *vpi_register_cb* se pak tyto informace zaregistrují.

6.5.3 Synchronizační funkce

V rozhraní kosimulace pro Verilog má synchronizační funkce podobu callback funkce, která reaguje na změny hodinového signálu. Callback funkce má jeden argument datového typu *p_cb_data*, což je ukazatel na strukturu *s_cb_data* zmíněnou výše, který obsahuje mimo jiné informaci o nové hodnotě hlídané proměnné. Na začátku funkce se tedy přečte tato hodnota, a pokud odpovídá logické hodnotě 1, tak se provede synchronizace. Jakmile se ukončí provádění kroku v simulaci Codasip, tak se provede funkce pro přenos změn výstupních a vstupně-výstupních portů simulovaného hardwaru.

6.5.4 Funkce pro přenos dat

Všechny vstupní a vstupně-výstupní porty budou obsluhovány každý vlastní callback funkcí. Jak již bylo zmíněno u synchronizační funkce, callback funkce v parametru předává novou hodnotu portu. Ve funkce *calltf* je jako reprezentace hodnoty vybrán řetězec binárních čísel. Porty jsou na

straně Verilogu reprezentovány pomocí datového typu *wire*. Proměnné tohoto datového typu mohou nabývat hodnot, které jsou v binárním řetězci reprezentovány následujícími znaky:

- '0': logická hodnota 0,
- '1': logická hodnota 1,
- 'z': stav vysoké impedance,
- 'x': neznámá hodnot – není možné zjistit hodnotu.

Znaky 'x' a 'z' budou převedeny na znak '0', aby se řetězec mohl převést na datový typ *codasip_int*. Po převedení už se hodnota pouze zapíše na port na straně Codasipu.

Pro opačný směr, tedy přenos hodnot výstupních a vstupně-výstupních portů Codasipu do Verilogu bude vytvořena jedna funkce, ve které se najednou přenesou všechny hodnoty všech těchto portů. Pro každý port se přečte hodnota typu *codasip_int*, převede se na řetězec, z řetězce se vybere jen tolik znaků, kolik má daný port bitů, a řetězec se pomocí funkce rozhraní PLI zapíše na stranu Verilogu.

6.5.5 Registrace nové uživatelské systémové úlohy

Při tvorbě rozhraní pro kosimulace s Verilogem je zapotřebí implementovat ještě registrační funkci nově definované uživatelské systémové úlohy. V registrační funkce je potřeba pouze naplnit strukturu *s_vpi_systf_data*. Tato struktura musí obsahovat informace o typu registrace (zda jde o systémovou úlohu nebo funkci), o názvu nové úlohy a o názvech funkcí *calltf* a *compiletf*. Struktura má ještě další položky, ale z pohledu tvorby kosimulačního rozhraní jsou bezvýznamné. Naplněná struktura se nakonec předá jako parametr funkci rozhraní PLI *vpi_register_systf*, která informace o nové systémové úloze zaregistruje na straně Verilogu.

Dále je nutné upozornit stranu Verilogu, že se má tato registrační funkce zavolat před samotným začátkem simulace. K tomu slouží seznam funkcí, které se mají zavolat při předzpracování simulace, s názvem *vlog_startup_routines*. Simulační platforma Verilog se při načtení sdílené knihovny snaží tento seznam v ní najít a vykonat funkce ze seznamu.

6.6 Kosimulace simulačních platforem Codasip a SystemVerilog

Rozhraní SystemVerilog DPI nenabízí tak jako předchozí dvě rozhraní tolik volnosti v přístupu k vnitřním strukturám simulátoru. Celé kosimulační rozhraní s touto platformou je tedy tvořeno voláním importovaných funkcí v kódu SystemVerilogu. Funkce pro inicializaci a pro

synchronizaci nemají v případě tohoto kosimulačního rozhraní žádný zvláštní kód navíc, proto je zde ani nebudu popisovat.

6.6.1 Zdrojový soubor SystemVerilogu

Zdrojový soubor pro SystemVerilog je velmi podobný jako zdrojový soubor pro Verilog. Opět v něm jsou dva moduly a mají stejné porty a proměnné jako u Verilogu. Rozdílem je, že jako datový typ portů jsem zvolil typ *logic* a ne *wire*. Je to z toho důvodu, že u portů datového typu *wire* se nedá měnit hodnota v blocích *always*, což při tvorbě kosimulačního rozhraní pro SystemVerilog je potřeba. Dalším problémem je, že do vstupně-výstupních portů se nedá přímo přiřadit hodnota. Proto jsem pro každý vstupně-výstupní port vytvořil v modulu dvě pomocné proměnné datového typu *logic*, jednu pro zápis do portu a jednu pro čtení z portu. Tyto proměnné jsou v kódu přiřazeny k portu pomocí příkazu *assign*. Dál v kódu už se tedy nepoužívá pro čtení a zápis do vstupně-výstupních portů samotné porty, ale tyto proměnné.

Do modulu popisujícího rozhraní simulovaného hardwaru oproti Verilogu přibude deklarace importovaných funkcí. Všechny vstupní a výstupní parametry funkcí jsou datového typu *logic* stejně jako porty kromě parametru inicializační funkce, který má typ *string*. V bloku *initial* se zavolá importovaná inicializační funkce kosimulačního rozhraní. V bloku *always*, který je asociován s nástupnou hranou hodinového signálu, se vyvolá jednak synchronizační funkce a hned za ní funkce pro přenos hodnot výstupních a vstupně-výstupních portů z Codasipu do SystemVerilogu, která má jako výstupní parametry všechny tyto porty. Dále pak pro každý vstupní a vstupně-výstupní port je vytvořen blok *always* asociovaný se změnou daného portu a v něm je volána funkce pro přenos hodnot tohoto signálu do Codasipu. Jako vstupní parametr je zadán asociovaný port.

6.6.2 Funkce pro přenos dat

Datový typ *logic* jazyku SystemVerilog převádí rozhraní DPI na datový typ *svLogic* a pole typu *logic* na typ *svLogicVecVal*. Proměnné datového typu *logic* mohou nabývat následujících hodnot:

- '0': logická hodnota 0,
- '1': logická hodnota 1,
- 'z': stav vysoké impedance,
- 'x': neznámá hodnot – není možné zjistit hodnotu.

Datový typ *svLogic* převádí tyto hodnoty na celočíselný typ *int*, kde '0' odpovídá číslu 0, '1' číslu 1, 'z' číslu 2 a 'x' číslu 3. Pro převod na datový typ *codasip_int* se číslo 1 nechává beze změny a čísla 2 a 3 se převedou na 0.

Funkce, která zapisuje hodnoty do výstupních a vstupně-výstupních portů modulu SystemVerilogu, má jako parametry tyto porty. Jelikož na straně SystemVerilogu jsou tyto parametry označeny jako výstupní, tak do funkce jazyka C jsou předávány odkazem. Ve funkci se pak pro každý port přečte hodnota portu na straně Codasipu, převede se na typ *svLogic* nebo *svLogicVecVal* a zapíše se do výstupního argumentu.

Pro každý vstupní a vstupně-výstupní port je vytvořena samostatná importovaná funkce, která má jako argument příslušný port. Jelikož je na straně SystemVerilogu parametr definován jako vstupní, tak je do funkce předáván hodnotou a ne odkazem. V této funkci se hodnota nejprve převede na řetězec, z řetězce se vytvoří proměnná datového typu *codasip_int* a ta se zapíše na příslušný port na straně Codasipu.

6.7 Kosimulace simulačních platforem Codasip a Matlab

Kosimulace s Matlabem je rozdílná od předchozích a to hlavně v tvorbě zdrojového kódu Matlabu. Nejsou zde žádné porty ani signály, na které by se dala kosimulace napojit a Matlab nenabízí žádné rozhraní, jak se napojit na datové struktury simulace. Implementace je tím pádem poněkud jednodušší než v předchozích rozhraních, protože je napsaná čistě v jazyce C bez žádných speciálních funkcí cizího rozhraní. Na druhou stranu je pak používání tohoto rozhraní pro uživatele nejsložitější. Stejně jako u DPI i zde nemají funkce pro inicializaci a synchronizaci žádnou přidanou funkčnost, proto zde popíši pouze zdrojový kód Matlabu a funkce pro přenos dat.

6.7.1 Zdrojový soubor Matlabu

K načtení sdílené knihovny pro kosimulaci se používá funkce *loadlibrary*. Jako parametry je nutné předat název knihovny a název hlavičkového souboru jazyka C. V tomto souboru musí být prototypy všech funkcí, které chceme v kódu Matlabu používat. Pro samotné zavolání nějaké funkce ze sdílené knihovny se využije funkce *callib*. Tato funkce přijímá jako parametry, název knihovny, název funkce a seznam parametrů volání této funkce.

Jak bylo popsáno v návrhu, samotná simulace probíhá v cyklu, ve kterém je možno číst z portů a zapisovat na porty simulovaného hardwaru. Hodnoty portů jsou na straně Matlabu reprezentovány pomocí řetězce. Dále je už na uživateli, jak s těmito hodnotami bude pracovat. Pokud chce převést hodnoty portů na číslo, nemůže použít základní číselné datové typy Matlabu, protože ty nedokážou pracovat s čísly o bitové délce 1024 bitů (maximální počet bitů portu). K převodu na číslo je možné použít například volně dostupnou knihovnu pro Matlab *VariablePrecisionInteger* [12], která je licencována licencí BSD. Tato knihovna dokáže pracovat s celými čísly neomezené délky. Druhou možností je použít *Symbolic Math Toolbox* [13], což je speciální toolbox pro Matlab. Tento nástroj

umožňuje pracovat také s čísly neomezené délky a to nejen s celočíselnými, ale i desetinnými. Pro potřeby práce s porty ale desetinná čísla nejsou potřeba.

6.7.2 Funkce pro přenos dat

Přenos dat bude reprezentován dvěma funkcemi – jedna funkce pro zápis do portu a druhá pro čtení z portu. Funkce pro zápis do portu přijímá dva řetězcové argumenty – jméno portu a zapisovanou hodnotu. Ve funkci se kontroluje, zda zadané jméno portu je skutečně vstupní nebo vstupně-výstupní port. Z toho důvodu se při generování zdrojového kódu vytvoří 3 globální konstantní pole názvů všech tří druhů portů. S položkami těchto polí se pak argument porovnává. Poté už jen následuje převod řetězce na datový typ *codasip_int* a zápis na daný port. Funkce pro čtení z portu má jako argument pouze jméno portu a hodnotu portu vrací jako návratovou hodnotu funkce. Zase se pouze zkontroluje, zda se název portu nachází v poli se seznamem výstupních nebo vstupně-výstupních portů, pak se přečte hodnota z portu, převede se na řetězec a použije se jako návratová hodnota.

7 Testování kosimulačních rozhraní

Po implementaci kosimulačních rozhraní pro simulační platformy VHDL, Verilog, SystemVerilog a Matlab jsem přešel k poslednímu bodu zadání, totiž k testování implementovaného řešení. Kosimulaci s platformami VHDL, Verilog a SystemVerilog jsem prováděl pomocí nástroje Modelsim.

Jako simulační příklad jsem použil předávání dat mezi simulačními platformami. Na straně Codasipu jsem použil již vytvořený model jádra procesoru s názvem *Codea2*. Následující porty procesoru jsem využil při kosimulaci:

- Vstupní 16-bitový datový port,
- Výstupní 16-bitový datový port,
- Výstupní 1-bitový port pro povolení zápisu na vstupní datový port,
- Výstupní 1-bitový port pro povolení čtení ze vstupního datového portu.

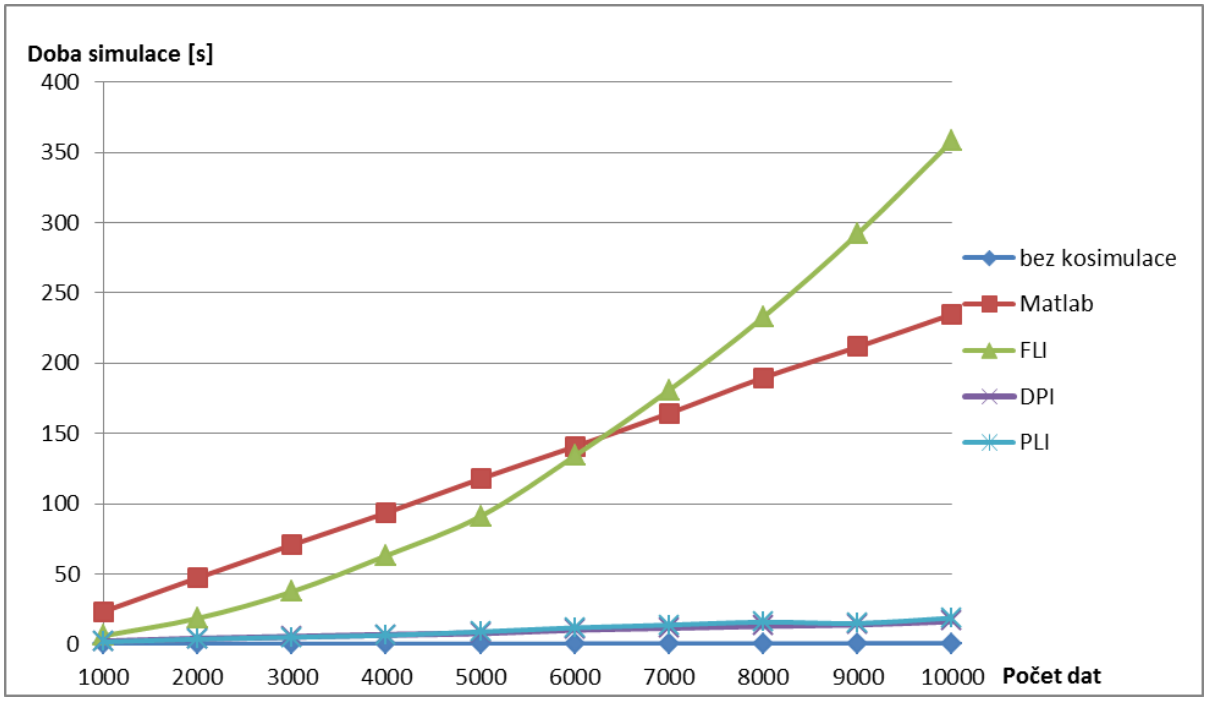
Aplikace spouštěná na tomto procesoru obsahovala cyklus s počtem iterací daným počtem přenášených hodnot. V každé iteraci se pak pouze přečetla data ze vstupního datového portu a hned přeposlala na výstupní datový port.

Na straně cizí simulační platformy byl implementován program, který reaguje na povolovací porty a podle toho, který z nich byl aktivován buď zapíše data na vstupní datový port, nebo data přečte z výstupního datového portu. Jako zdroj dat jsem použil u kosimulace s Verilogem, SystemVerilogem a Matlabem soubor, ve kterém na každém řádku byla jedna hodnota. Jazyk VHDL nenabízí jednoduchou možnost práce se soubory, proto jsem v tomto případě zapsal data přímo do kódu ve formě konstantního pole. Simulace jsem prováděl i bez použití kosimulačního rozhraní. V tomto případě se o zápis a čtení hodnot starala sdílená knihovna vytvořená v jazyce C++, která byla přilinkována k simulátoru Codasip.

Simulaci jsem spouštěl pro počet hodnot v rozmezí od 1000 do 10000 s krokem 1000. Pro každý počet hodnot jsem provedl 10 simulací a výsledné hodnoty grafu jsou dány jejich průměrem. Simulační platforma Codasip pro každou simulaci vypisuje na výstup dobu provádění simulace, počet provedených cyklů v simulaci a rychlost provádění simulace. Poslední údaj je dopočítaný jako podíl počtu provedených cyklů a doby provádění simulace. Z výsledků jsem sestrojil graf popisující závislost doby provádění simulace na počtu přenesených hodnot a graf závislosti rychlosti simulace taktéž na počtu přenesených hodnot.

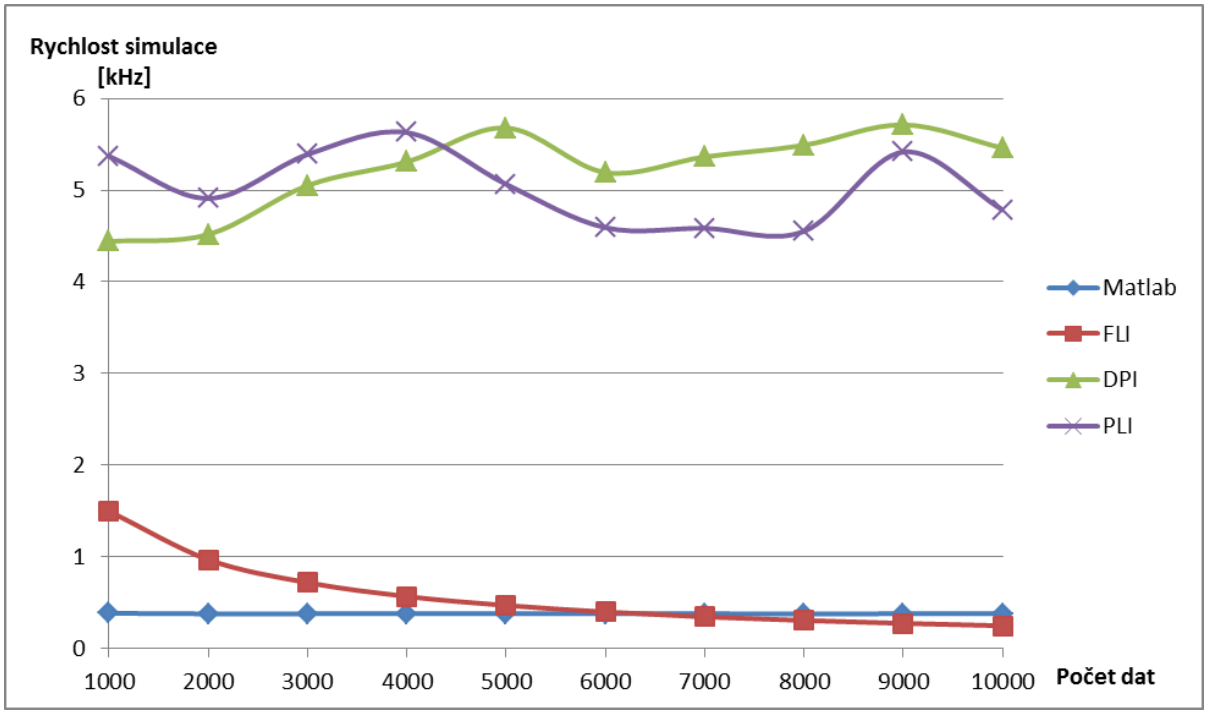
Simulaci jsem prováděl na počítači s touto konfigurací:

- Procesor: AMD Turion 64 X2 TL-60 2.00 GHz
- Frekvence FSB: 200MHz
- Operační paměť: 2.00 GB
- Operační systém: Ubuntu 11.10 64-bitová verze



Graf 1:Závislost doby provádění simulace na počtu přenesených hodnot

Nejrychleji samozřejmě probíhala simulace bez použití jakéhokoli rozhraní. Doba provádění se zvyšovala jen velmi mírně a nepřekročila 1s. Pro kosimulaci s použitím rozhraní DPI a PLI byly výsledky téměř podobné a lišily se maximálně v desetinách sekundy. Také zde se doba provádění zvyšovala velmi mírně s přibývajícím počtem přenesených hodnot. Kosimulace s Matlabem už znamenala znatelnější zvýšení doby simulace, ovšem při každém zvýšení počtu přenesených hodnot se doba provádění zvýšila o konstantní přírůstek. Jiné jsou ovšem výsledky při použití rozhraní FLI. Pro velmi malý počet hodnot je doba simulace téměř stejná jako u rozhraní DPI a PLI. Při zvyšování počtu hodnot doba simulace neroste lineárně, ale exponenciálně. Pro toho rozhraní jsem vyzkoušel ještě jednu simulaci s počtem přenesených hodnot rovným 20000, abych zjistil, zda exponenciální růst bude stále pokračovat. Doba simulace v tomto případě byla téměř 1400s, z toho vyplývá, že růst se zachoval.



Graf 2: Závislost rychlosti simulace na počtu přenesených hodnot

Další graf pouze potvrzuje závěry z předchozího grafu. Při kosimulaci s použitým rozhraní DPI a PLI je rychlost téměř totožná a pohybuje vysoko nad rychlostí kosimulace s Matlabem a s rozhraním FLI. Navíc je zde vidět, že rychlost kosimulace s Matlabem je velmi stabilní, téměř nereaguje na počet přenášených hodnot.

8 Závěr

Hlavním účelem této diplomové práce bylo vytvořit kosimulační rozhraní pro nástroj pro HW/SW co-design Codasip, který je vyvíjen v rámci projektu Lissom. V rámci této práce byly zhodnoceny možnosti kosimulace se simulačními platformami jazyků pro popis hardwaru a s nástrojem Matlab. Z analýzy jednotlivých externích rozhraní simulačních platforem vznikl návrh a implementace kosimulačního rozhraní mezi simulační platformou Codasip a platformami VHDL, Verilog, SystemVerilog a Matlab.

Návrh a implementace kosimulačního rozhraní byly vytvářeny s ohledem na co nejjednodušší používání rozhraní pro uživatele. Při generování konkrétního simulátoru Codasip s rozhraním pro kosimulaci s cizí simulační platformou je automaticky vygenerován i zdrojový kód v cizím jazyce, který toto rozhraní používá, aby jej uživatel nemusel sám vytvářet. Tím pádem tedy ani nemusí znát detaily fungování rozhraní.

Ze závěrečného testování vyplývá použitelnost jednotlivých rozhraní. Zatímco kosimulace pomocí rozhraní PLI simulační platformy Verilog a rozhraní DPI simulační platformy SystemVerilog mají malý vliv na celkovou rychlost simulace, tak rozhraní sdílených knihoven pro Matlab a rozhraní FLI simulační platformy VHDL jsou při náročnějších simulacích kvůli režii jimi zanesené téměř nepoužitelné.

Mnou vytvořené rozhraní se bude dále používat v nástroji Codasip. Do nástroje přineslo rozhraní možnost nesimulovat samostatně pouze modelovaný procesor, ale simulovat chování procesoru při komunikaci s připojenými periferiemi, které jsou modelovány v jiných jazycích. Moje práce nabízí i možnosti pro budoucí vývoj. Určitě je možné postupně do nástroje Codasip přidávat nová rozhraní pro kosimulaci s jinými simulačními platformami, než pro které jsem je vytvořil já. Dále je možné ještě optimalizovat mé řešení a ještě tím zrychlit provádění kosimulace.

Práce na této diplomové práci byla opravdu zajímavá. Splnil jsem během ní všechny body zadání. Dozvěděl jsem se hlavně nové informace z oboru simulací a modelování hardwaru, jelikož s tímto oborem jsem se dříve setkával jen okrajově. Také jsem se naučil používat jazyky Verilog a SystemVerilog, které jsem do začátku mé práce neznal.

Literatura

[1] HÜBERT, Heiko . *Survey of HW/SW cosimulation techniques and tools* [online]. Stockholm, Švédsko, 1998 . Dostupné z: http://www.cs.indiana.edu/~bpisupat/work/papers/hubert98survey.pdf.

[2] Using Architectural Description Languages to Improve Software Quality and Correctness. *Itl.nist.gov* [online]. [cit. 2012-05-20]. Dostupné z: http://www.itl.nist.gov/div897/ctg/adl/adl_info.html

[3] CLEMENTS, Paul C. A Survey of Architecture Description Languages. In: *Proceedings of the 8th International Workshop on Software Specification and Design, March 22-23, 1996, Schloss Velen, Germany*. Los Alamitos: IEEE Computer Society Press, c1996, s. 16-25. ISBN 0-8186-7361-3.

[4] MEHTA, Gaurav. *Hardware Description Languages* [online]. University of California, Santa Barbara [cit. 2012-04-10]. Dostupné z: http://www.cs.ucsb.edu/~gaurav_mehta/reports/cs263.pdf

[5] COSPONSORS, SCC 20. *IEEE standard VHDL language reference manual* [online]. New York, N.Y: Institute of Electrical and Electronics Engineers, 2000 [cit. 2012-05-3]. ISBN 07-381-1949-0. Dostupné z: http://www.ece.uic.edu/~dutt/courses/ece368/lect-notes/VHDLref.pdf

[6] MODEL TECHNOLOGY INCORPORATED. *Modelsim FLI reference* [online]. Portland, USA, 2002. Dostupné z: http://homepages.cae.wisc.edu/~ece554/new_website/ToolDoc/Modelsim_docs/docs/pdf/fli.pdf.

[7] SUTHERLAND, Stuart. *Verilog® HDL Quick Reference Guide*: *based on the Verilog-2001 standard (IEEE Std 1364-2001)* [online]. 2001 [cit. 2012-04-25]. ISBN 1-930368-03-8. Dostupné z: http://www.sutherland-hdl.com/online_verilog_ref_guide/verilog_2001_ref_guide.pdf

[8] SUTHERLAND, Stuart. *The Verilog PLI handbook*: *a user's guide and comprehensive reference on the Verilog programming language interface*. 2nd ed. Norwell: Kluwer Academic Publishers, 2002, 784 s. ISBN 07-923-7658-7.

[9] ACCELLERA ORGANIZATION, Inc. *Accellera SystemVerilog 3.1*: *Language Reference Manual*. 2003. Dostupné z: http://www.eda.org/sv/SystemVerilog_3.1a.pdf.

[10] MATLAB: The Language of Technical Computing. *Mathworks.com* [online]. [cit. 2012-05-10]. Dostupné z: http://www.mathworks.com/products/matlab/

[11] External Interfaces. *Mathworks.com* [online]. [cit. 2011-11-20]. Dostupné z: http://www.mathworks.com/help/techdoc/matlab_external/bp_kqh7.html.

[12] Variable Precision Integer Arithmetic. *Mathworks.com* [online]. [cit. 2012-05-3]. Dostupné z: http://www.mathworks.com/matlabcentral/fileexchange/22725

[13] Symbolic Math Toolbox. *Mathworks.com* [online]. [cit. 2012-05-3]. Dostupné z: http://www.mathworks.com/products/symbolic/?s_cid=learnmore_%20doc

Seznam příloh

Příloha 2

Generovaný kód VHDL

```
library ieee;
use ieee.std_logic_1164.all;


entity fli_model is
	port(
		clk: in std_logic;
		rst: in std_logic;
		port_out: out std_logic_vector(7 downto 0);
		port_out_en: inout std_logic;
		port_in: in std_logic_vector(7 downto 0);
		port_in_en: inout std_logic);
end fli_model;
architecture arch of fli_model is
	attribute foreign of arch : architecture is "initForeign libfli_spp.so;intersim2 -i spp.xexe -x
	simout.xml -n spp;";
begin
end arch;



library ieee;
use ieee.std_logic_1164.all;


entity top is
end top;
architecture arch of top is
	component fli_model
		port(
			clk: in std_logic;
			rst: in std_logic;
			port_out: out std_logic_vector(7 downto 0);
			port_out_en: inout std_logic;
			port_in: in std_logic_vector(7 downto 0);
			port_in_en: inout std_logic);
	end component;
	constant clk_period : time := 10 ns;
	signal end_of_sim : boolean := false;
	signal clk: std_logic := '1';
	signal rst: std_logic := '1';
	signal port_out: std_logic_vector(7 downto 0);
	signal port_out_en: std_logic;
	signal port_in: std_logic_vector(7 downto 0);
	signal port_in_en: std_logic;
for comp : fli_model use entity work.fli_model(arch);
```

```
begin
        comp : fli_model
        port_map (
                clk => clk,
                rst => rst,
                port_out => port_out,
                port_out_en => port_out_en,
                port_in => port_in,
                port_in_en => port_in_en);

        clk_process : process
        begin
                if end_of_sim = false then
                        clk <= '1';
                        wait for clk_period/2;
                        clk <= '0';
                        wait for clk_period/2;
                else
                        clk <= '0';
                        wait;
                end if;
        end process;

        main : process
        begin
                rst <= '0';
                wait for clk_period * 10;
                rst <= '1';
                wait for clk_period * 1000;
                end_of_sim <= true;
                wait;
        end process;
end arch;
```

Příloha 3

Generovaný kód Verilogu

```
module pli(
	input wire clk,
	input wire rst,
	output wire [7:0] port_out,
	inout wire port_out_en,
	input wire [7:0] port_in,
	inout wire port_in_en);

	initial
	begin
		$foreign(pli, "intersim2 -n spp -x simout.xml -i spp.xexe");
	end

endmodule

module top();
	reg clk = 1'b0;
	reg rst = 1'b1;
	wire [7:0] port_out;
	wire port_out_en;
	reg [7:0] port_in;
	wire port_in_en;

	always
	begin
		#5 clk = 1'b1;
		#5 clk = 1'b0;
	end

	pli foreign_pli(
		.clk(clk),
		.rst(rst),
		.port_out(port_out),
		.port_out_en(port_out_en),
		.port_in(port_in),
		.port_in_en(port_in_en));

endmodule
```

Příloha 4

Generovaný kód SystemVerilogu

```
module dpi(
	input logic clk,
	input logic rst,
	output logic [7:0] port_out,
	inout logic port_out_en,
	input logic [7:0] port_in,
	inout logic port_in_en);

	import "DPI-C" task initForeign(input string param);
	import "DPI-C" task clk_process();
	import "DPI-C" task rst_process();
	import "DPI-C" task read_ports(output logic [7:0] port_out, output logic port_out_en,
		output logic port_in_en);
	import "DPI-C" task write_port_out_en(input logic port_out_en);
	import "DPI-C" task write_port_in(input logic [7:0] port_in);
	import "DPI-C" task write_port_in_en(input logic port_in_en);

	logic port_out_en_out;
	logic port_out_en_in;
	assign port_out_en = port_out_en_out;
	assign port_out_en_in = port_out_en;
	logic port_in_en_out;
	logic port_in_en_in;
	assign port_in_en = port_in_en_out;
	assign port_in_en_in = port_in_en;

	always @(posedge clk)
	begin
		clk_process();
		read_ports(port_out, port_out_en_out, port_in_en_out);
	end

	always @(negedge rst)
	begin
		rst_process();
	end

	always @(port_out_en_in)
	begin
		write_port_out_en(port_out_en_in);
	end

	always @(port_in)
	begin
		write_port_in(port_in);
	end
```

```
        always @(port_in_en_in)
        begin
                write_port_in_en(port_in_en_in);
        end

        initial
        begin
                initForeign("intersim2 -n spp -x simout.xml -i spp.xexe");
        end

endmodule

module top();
        logic clk = 0;
        logic rst = 1;
        logic [7:0] port_out;
        wire port_out_en;
        logic [7:0] port_in;
        wire port_in_en;

        always
        begin
                #5 clk = 1;
                #5 clk = 0;
        end

        dpi foreign(
                .clk(clk),
                .rst(rst),
                .port_out(port_out),
                .port_out_en(port_out_en),
                .port_in(port_in),
                .port_in_en(port_in_en));

endmodule
```

Příloha 5

Generovaný kód Matlabu

```
loadlibrary libmatlab_spp matlab_spp.h alias lib;
calllib('lib', 'initForeign', 'intersim2 -i spp.xexe -x simout.xml -n spp');
end_of_sim = calllib('lib', 'end_of_sim');
while end_of_sim == 0
        calllib('lib', 'clk');
        end_of_sim = calllib('lib', 'end_of_sim');
end
unloadlibrary lib;
```